

AFRL-IF-WP-TR-2001-1547

**TRUSTWORTHY SOFTWARE: WHEN
COMPUTERS SERVE AS PROXIES FOR
HUMANS**

DR. TIM SHEARD

**OREGON GRADUATE INSTITUTE
PACIFIC SOFTWARE RESEARCH CENTER
P.O. BOX 91000
PORTLAND, OR 97291-1000**



AUGUST 2000

FINAL REPORT FOR PERIOD 28 SEPTEMBER 1999 – 21 AUGUST 2000

Approved for public release; distribution unlimited

**INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334**

20020103 133

NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

Contract: F33615-99-C-1511

Contractor: Oregon Graduate Institute

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

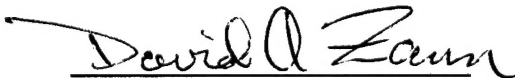
THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.



RONALD SZKODNY
Program Manager
Advanced Architecture & Integration Branch



JOHN C. OSTGAARD
Team Leader
Advanced Architecture & Integration Branch



DAVID A. ZANN, Chief
Advanced Architecture & Integration Branch
Information Systems Division

This report is published in the interest of scientific and technical information exchange and does not constitute approval or disapproval of its ideas or findings.

Do not return copies of this report unless contractual obligations or notice on a specific document require its return.

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> <i>OMB No. 074-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE AUGUST 2000	3. REPORT TYPE AND DATES COVERED Final, 09/28/1999 – 08/21/2000	
4. TITLE AND SUBTITLE TRUSTWORTHY SOFTWARE: WHEN COMPUTERS SERVE AS PROXIES FOR HUMANS			5. FUNDING NUMBERS C: F33615-99-C-1511 PE: 62301E PN: ARPI TN: FS WU: 0D	
6. AUTHOR(S) DR. TIM SHEARD				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) OREGON GRADUATE INSTITUTE PACIFIC SOFTWARE RESEARCH CENTER P.O. BOX 91000 PORTLAND, OR 97291-1000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) INFORMATION DIRECTORATE AIR FORCE RESEARCH LABORATORY AIR FORCE MATERIEL COMMAND WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334 POC: Ronald Szkody, AFRL/IFSC, (937) 255-4709 x4165			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-WP-TR-2001-1547	
11. SUPPLEMENTARY NOTES: This is a DARPA-funded effort under the software enabled control program.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) In this effort, we propose to study the feasibility of rebuilding the infrastructure of embedded system software from the ground up, with trustworthiness as a fundamental premise. Specifically, we propose to develop a collection of both domain-specific and general purpose abstract machines, languages and tools that have built-in formal methods, and develop new ones based on sound mathematical principles. We will treat hardware, operating systems, networking, and programming languages with the same methodology, ensuring their seamless integration with respect to formal system properties. We propose to develop these ideas into a prototype proof-of-concept computing system built from the ground up with the concern for trustworthiness.				
14. SUBJECT TERMS Reflective meta-programming environments for Haskell, Facets of Trust, Functional Reactive Programming Model LINDA shared virtual memory				15. NUMBER OF PAGES 114
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR	
NSN 7540-01-280-5500			Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102	

Table of Contents

<u>Section</u>	<u>Page No.</u>
Preface	iv
ITO 2000 Project Summary Collection	5
Appendix A: Functional Reactive Programming from First Principles.....	11
Appendix B: DALI: An Untyped, CBV Functional Language Supporting First-Order Datatypes with Binders (Summary).....	22
Appendix C: DALI: An Untyped CBV Operational Semantics and Equational Theory for Datatypes with Binders (Technical Development).....	34
Appendix D: Constructing Modular Type Systems	69
Appendix E: Frappe: Functional Reactive Programming in Java*	94

Preface

This effort was initiated under a DARPA/ITO Broad Agency Announcement #99-07 in the topic area of Trusted Computing under a Information Technology Expedition search. The proposal, named "Trustworthy Software: When Computers Serve as Proxies for Humans" was initially funded under the Software Enabled Control (SEC) program as contract F33615-99-C-1511 in September 1999. Funding was moved to the Program Composition for Embedded Systems (PCES) program in FY2000. The contract was terminated in August 2000 when it became apparent that the products were incompatible with the PCES efforts. Insufficient progress was accomplished to generate a complete final report, so a summary of work attempted referenced to the Statement of Work is being published as the final report along with it's ITO Program Summary for FY2000. The appendix contains five papers published by OGI faculty and students related to this effort.

Trustworthy Software: When Computers Serve as Proxies for Humans

Final Project Report
September 28, 1999 through August 21, 2000

Jan 15, 2001

AFRL/IFSC
Attn: Ron Szkody
2241 Avionics Circle
WPAFB, OH 45433-7318

Re: Contract Number F33615-99-C-1511

Dear Mr. Szkody:

The following is the Final Report on "Trustworthy Software: When Computers Serve as Proxies for Humans" concentrating on the period from March 31, 2000, until the termination of the project on August 21, 2000.

Tasks:

3.1 – Build a reflective meta-programming system for Haskell. This tool has two purposes. First, it supports the construction of staged programs that have good performance properties, and second it provides a high-level tool for the construction of formal property analyses in the same language as the programming environment. This tool shall be used to construct a program along the lines of the HOL-prover the contractor has built in MetaML. (CDRL Data Item #s A007 Software user manual, A008, software design description, and A004, interim reports)

Completed:

- Designed a flexible datatype representation for Haskell programs.
- Completed a parser that parses the complete Haskell Language, and produces an algebraic datatype that represents Haskell programs.
- Completed design and implementation of a flexible type checker.

Not Completed:

- Interfaces to Haskell's class system not implemented.
- A007, Software user manual.
- A008, Software design description.

3.2 – Develop facets of trust. The contractor shall concentrate on exploiting formal properties of programs and accountability issues. Other facets shall be developed as needed. For example, the contractor shall develop a theory of accountability to formalize tracking the dependency of answers on contributed inputs and allow for the delegation of accountability. The contractor shall also include a notion of adaptation. Whereby a system can identify when a problem occurs, hold someone else accountable, then adapt by exploiting that accountability. (CDRL Data Item # A004, interim reports)

Completed:

- Developed analysis that guarantee noninterference between processes that utilize separate partitions.
- Began investigating model-based synthesis of trustworthy software by using Functional Reactive Programming (FRP) as a model for interactive systems.
- Completed a design and implementation of FRP that runs on top of the Java Beans event based model.

Not Completed:

- Using Meta-Haskell tool to build an automatic translator to do the job, which was done by hand in the FRP implementation above.

3.3 – Apply the tools and techniques of meta-programming, higher-order functional abstraction, and advanced type systems to represent and analyze facets of trust. This task shall investigate the following areas:

3.3.1 – Develop encodings of important properties that are directly checkable by an extended Haskell type system.

Completed:

- Built generic framework for building Hindley-Milner plus constraint type systems, which can be reused in many contexts.
- Implemented a for Haskell, that was designed to be extended.
- Devised a new method for representing programs that use the type system of the meta-language in a new and unique way that enforces the scoping discipline of the object language.

3.3.2 Exploit monads to track the presence and absence of effects capabilities

No work on sub-task completed.

3.3.3 Use an extended Hindley-Milner type system such as the HMX system to track the correct use of physical “units” or other domain-specific properties of interest.

No work on sub-task completed.

3.3.4 Build an HOL-like theorem prover to track various levels of ‘correctness’, some of which might be applicable only to specific domains of interest, for example, termination.

No work on sub-task completed.

3.4 – Build a prototype, which demonstrates the technology within a concrete scenario. The scenario shall be both realistic and well developed within a rich domain of interest such as robotics or multi-sensor control systems. The contractor shall demonstrate the synthesis of trustworthy behaviors in the presence of untrustworthy information. The system shall include multiple sensors and agents of varying trustworthiness. The contractor shall also use meta-programming and dynamic functions to modify system functionality in a trustworthy manner as the system is running. (CDRL Data Item #s A007 Software user manual, A008, software design description, and A004, interim reports)

Completed:

- Developed a model for interactive systems based on Functional Reactive Programming.
- Implemented parallel interactive systems using Haskell systems communicating via the Linda shared virtual memory.
- Developed a method for creating Java programs (using Java Beans) from FRP-based models.

Not Completed:

- Demonstration within a concrete scenario.
- A007, Software user manual.
- A008, Software design description.

THIS PAGE IS INTENTIONALLY LEFT BLANK

ITO 2000

Project Summary Collection

Verify a Technical Report

Attention: Your report has not been accepted yet!

Please review your report for accuracy, grammar and spelling errors. If you are satisfied with your report then click the "Submit" button at the bottom of this page. If you wish to make changes to your report then click the "Back" button on your browser.

Project Title:	Trustworthy Software: when Computers Serve as Proxies for Humans
Organization:	Oregon Graduate Institute
AO Number:	H826
Contract Number:	F33615-99-C-1511
Start Date:	30 SEPT 1999
End Date:	30 SEPT 2002
Principal Investigator	
Name: Tim Sheard	
Address: Pacific Software Research Center	
20000 NW Walker Road	
City, State Zip: Beaverton, OR 97006-8921	
Phone: 503-748-1439	
Fax: 503-748-1553	
Email: sheard@cse.ogi.edu	
Level Of Participation - Billed: 30%	
Level Of Participation - Unbilled: 0%	
Financial POC	

Name: Deborah Golden-Eppelein
 Address: Oregon Graduate Institute
 20000 NW Walker Road
 City, State Zip: Beaverton, OR 97006-8921
 Phone: 503-748-1031
 Fax: 503-748-1387
 Email: dgolden@admin.ogi.edu

Project URL:	http://www.cse.ogi.edu/pacsoft/projects/TW/Default.htm
Objective:	To build and demonstrate a set of modern tools to ensure that tomorrow's software meets standards of trustworthiness and reliability.
Approach:	<p>Trust is an informal idea that is part of a complicated metaphor. Trust is built, at some level, on faith. No one can completely trust any entity; yet (some) people make reliable decisions about who and what they can trust everyday. Surely software can do the same. A useful approach to the construction of trustworthy software is in part based upon the the following observation: <i>Trust depends on Understanding</i>.</p> <ul style="list-style-type: none"> • Use a succinct and precise definition of a complex system to achieve trust. • A model describes a complete system, encompassing all elements (components or processes) at once. • Different models focus on different facets of trust. <p>Models. A model captures the meaning (semantics) of a system in a declarative and composable way. The modeling language should represent the system in a way that promotes human understanding. An important way to accomplish this is to use the language of the underlying domain embedded in a domain specific language. This allows the inheritance of methods of formal analysis from existing known domains.</p> <p>Strategies. We have used the following strategies to build reliable, trustworthy systems:</p> <ul style="list-style-type: none"> • Modeling. Trust by understanding. An abstraction prevents the implementation details from getting in the way of what it does, but provides a clean model to the user of how it behaves. When a component has an internal model of its interaction with other components it can use this model to make informed decisions on what and how to trust. • Composition. Large systems can only be built in an effective manner by composing them from smaller understood sub-components. Compositionality is the key to scaling. The meaning of a component must depend only on the meaning of its sub-components to minimize unintended component interaction.

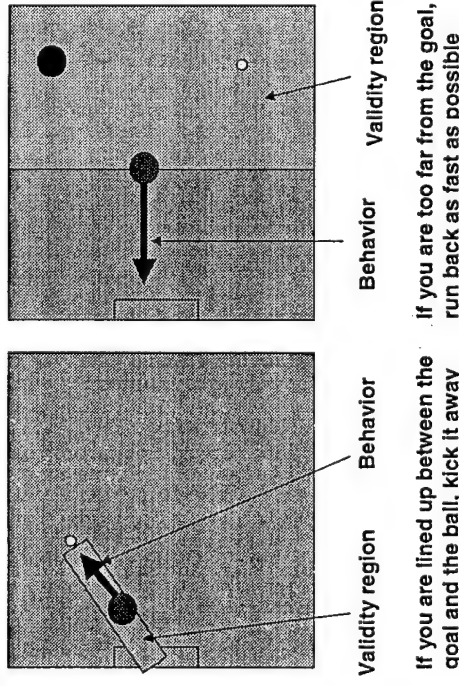
	<p>The use of higher-order languages makes compositionality possible.</p> <ul style="list-style-type: none"> • Factoring. Separating concerns of functionality from concerns of accountability and trust. By using lifting operators that apply trust policies to functional components, trust is achieved in an orthogonal way from functionality. • Generation. Trust is Ensured by the generation of components from abstract models where correctness is by observation. To separate functional concerns from concerns of trust, a "weaving" of separate specifications is necessary. This weaving is complex, and usually beyond the capability of humans to perform on large systems. Generation technology provides the tool to accomplish this. Generation technology can be made generic, and can be reused over the life of many systems. The generator must implement a faithful translation. In order to ensure that all properties of the model are maintained, It is necessary to reason about the generated code, only by inspecting the code of the generator. Models of how generators work are a necessary part of trustworthy systems.
Recent Accomplishments:	<ul style="list-style-type: none"> • Meta-Programming semantics and implementation. We have implemented and released (March 2000) MetaML, a higher-order, typed programming languages that provides special support for the construction, manipulation, and execution of code. The release of the MetaML system can be found on the project web pages. <p>Meta-programming systems provide a common, reusable solution to a number difficult problems that all program generators implementations must provide. For a brief overview of the issues involved see our Taxonomy of Meta-Programming Systems manifesto.</p> <ul style="list-style-type: none"> • FRP Semantics. We have developed a precise semantic model for FRP. And continues to investigate the core set of FRP constructs. Such a core provides a reference implementation of FRP for other researchers. This work is detailed in the paper: <p>Functional Reactive Programming from First Principles, Zhanyong Wan and Paul Hudak . ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI), June 2000, ACM Press.</p> <ul style="list-style-type: none"> • Representing Object Programs. Programs are are data. But they are complex entities. How can programs be represented in way that hides unnecessary details, but make their important details easy to manipulate. We have developed a new method for representing programs with binding constructs. It is reported in the following paper: <p>DALI: An Untyped, CBV Functional Language Supporting First-Order Datatypes with Binders. Tim Sheard, Walid Taha and</p>

	<p>Emir Pasalic.</p> <ul style="list-style-type: none"> • Modular Type Systems. We have developed a mechanism for constructing type systems from modular components. The ideas are developed in the following paper: Constructing Modular Type Systems. Chiyan Chen. • The MetaHaskell System. We have been constructing MetaHaskell, a lazy meta-programming system for the Haskell language. Our progress to date includes a full parser, a generic type checker, a generic type checking monad (which include reusable components for generic unification, tracking errors to point of cause, failure recovery and backtracking). Our progress to date can be found on the project web pages. • Meta-programming: Restructuring parallel programs to increase throughput. Parallel Functional Reactive Programming, by J. Peterson, V. Trifonov, and A. Serjantov. Practical Aspects of Declarative Languages. (Jan. 17-18, 2000, Boston Massachusetts). • Efficient Compilation of FRP programs. We have investigated FRP as a way of expressing interaction in Java. This Integration allows existing Java components to interact with the FRP framework and support composition of these Java objects. This system is currently restricted to first-order FRP (no dynamic behavior creation). This work is described in the paper: "Frappe: Functional Reactive Programming in Java" by Antony Courtney.
Current Plan:	<p>Modelling Interaction: We have used Functional Reactive Programming (FRP) to model interactive systems. Examples we have built include control systems, vision, robotics, and reactive animation. FRP is a general framework for adding interaction and time dependent behavior in a modular and composable way.</p> <p>Research Areas Addressed in FRP:</p> <ul style="list-style-type: none"> • Essential FRP semantics. We have defined the meaning of interactive systems using FRP by giving FRP a precise denotational semantics. We have also showed how to reason and compare an actual implementation to its denotation. • Analysis of FRP-based systems. We have adapted existing analysis tools to handle interaction. We have used FRP to construct models in specific domains. For example, web servers, and robot control systems. • Compilation techniques for FRP-based systems. We have realized FRP-based systems practically and reliably using higher-order programming languages, program generation and meta-programming. <p>Meta-programming: Generating Solutions. We have used</p>

	<p>meta-programming as a generic tool for generating solutions and verifying properties from high-level specifications. Example generated solutions include: compilers for DSLs and staged pattern matching. Example properties include extended type systems, and generic property maintenance and propagation. We have found meta-programming to be a generic framework for manipulating programs in a high-level and semantically coherent way.</p> <p>Research Areas Addressed in Generation.</p> <ul style="list-style-type: none"> • Semantics. We have devised the first useful logic that can be used to compare whether two meta-programs compute the same thing. The logic can also be used to compare the equivalence of a program and its staged counterpart. • Type Systems. We have devised and implemented a type system for the meta-language MetaML that ensures safety properties of generated programs by typing the meta-programs themselves. • Representing Programs. Programs are data. But they are complex entities. We have devised a new method for representing programs in way that hides unnecessary details (such as the actual names of local variables) but make their important details easy to manipulate. • Modular Composition. We have structured language implementations in a way that is easy to reuse by applying the technologies of monads and staging. This makes it easier to build and prototype new language systems rapidly.
Technology Transition:	By making our systems public, we have encouraged other groups to use them as tools in their own work. The MetaML system has been publicly available since March 2000.
Comments / Questions / Anything else you need:	<p>The Trustworthy Project is a collaboration between the Oregon Graduate Institute and Yale University. The Trustworthy personnel include: Tim Sheard, principal investigator at OGI, and John Peterson principal investigator at Yale. Postdoc, Zino Benaissa, OGI (departing July 31, 2000), and Postdoc, Bill Harrison, OGI (just starting June 7, 2000). Students Chiyan Chen, OGI, Antony Courtney, Yale, and Zhanyong Wan, Yale.</p> <p>Please note that the above listed contract start date of 30 September, 1999 is incorrect. The correct start date should be 28, September 1999. Notification was sent to ito-pi-help@darpa.mil but no response has been received to date.</p>
<div style="border: 1px solid black; padding: 5px; display: inline-block;">Submit Technical Report</div>	

Trustworthy Software: When Computers Serve as Proxies for Humans

Control from the composition of independent behaviors depending on situation.



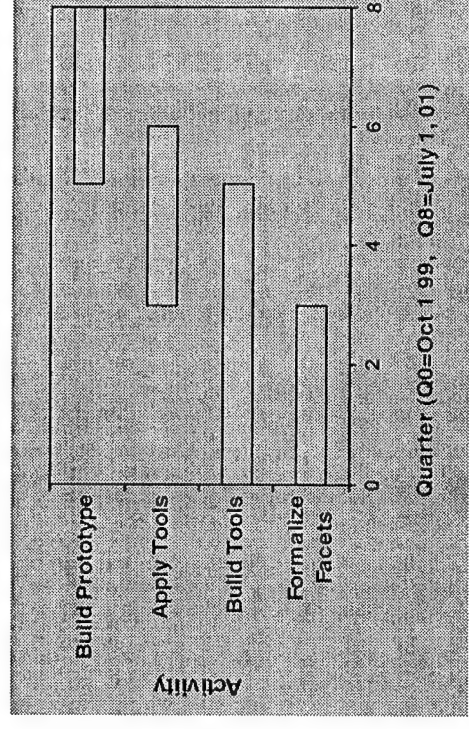
New ideas

- Separating concerns of functionality from concerns of accountability and trust.
- Compositionality = scaling. Meaning of a component must depend only on the meaning of its sub-components to minimize unintended component interaction
- Meta-programming is a generic framework for manipulating programs in a high-level and semantically coherent way.

Impact

- Integration of components of various sources to make components with predictable behavior.
- Systems that alter behavior depending on context (composition of control systems) and which commit to and execute complex parameterized behaviors (control systems) based on situation (planning, learning)

Schedule



Tim Sheard - Oregon Graduate Institute & John Peterson - Yale University

Functional Reactive Programming from First Principles

Zhanyong Wan
Yale University
Department of Computer Science
P.O. Box 208285
New Haven, Connecticut 06520
zhanyong.wan@yale.edu

Paul Hudak
Yale University
Department of Computer Science
P.O. Box 208285
New Haven, Connecticut 06520
paul.hudak@yale.edu

ABSTRACT

Functional Reactive Programming, or *FRP*, is a general framework for programming hybrid systems in a high-level, declarative manner. The key ideas in FRP are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are time-ordered sequences of discrete-time event occurrences. FRP is the essence of *Fran*, a domain-specific language embedded in Haskell for programming reactive animations, but FRP is now also being used in vision, robotics and other control systems applications.

In this paper we explore the formal semantics of FRP and how it relates to an implementation based on *streams* that represent (and therefore only approximate) continuous behaviors. We show that, in the limit as the sampling interval goes to zero, the implementation is faithful to the formal, continuous semantics, but only when certain constraints on behaviors are observed. We explore the nature of these constraints, which vary amongst the FRP primitives. Our results show both the power and limitations of this approach to language design and implementation. As an example of a limitation, we show that streams are incapable of representing instantaneous predicate events over behaviors.

1. INTRODUCTION

How does one show that a language implementation is correct? In the programming language research community, we normally do this by showing that the implementation is faithful, in some formal sense, to an abstract denotational or operational semantics of the language. Indeed, if all goes well, we can formally *derive* the implementation from the semantics. Such is the nature of “provably correct compilation.”

However, in the case of Functional Reactive Programming (FRP), a novel language involving continuous time-varying values as well as discrete events, the situation is not as clear, and several questions arise:

1. How does one express the formal semantics of FRP? We partially answered this question in a previous paper [7], and we refine that strategy here.
2. How does one implement continuous time-varying behaviors? In this paper we explore perhaps the most obvious technique, one based on *streams* that represent sampled behaviors (in a signal processing sense). However, this representation is only an approximation to the continuous values, which leads to the next question.
3. In what sense is an approximating stream-based implementation *correct* with respect to the formal semantics, and what are its limitations (for example, are there values that cannot be represented)? The interaction of these issues with the reactive component of FRP makes this especially interesting.

In this paper we provide answers to all of these questions. Specifically, we give a denotational semantics to FRP, and show that in the limit as the sampling interval goes to zero, a stream-based implementation corresponds precisely to the formal semantics, but only with suitable constraints on the nature of behaviors. The good news here is that most of the common things that we express with FRP programs are well behaved, and laws that we expect to hold in mathematics are justified, in the limit, when reasoning about FRP programs. For example, we can safely apply most FRP primitives, such as integration, to behaviors that are *discontinuous* (in a certain way to be described later), which is critically important given that the reactive component of FRP creates discontinuities quite often.

The bad news is that, since FRP is mathematically very rich, many ill behaved values can be expressed. As a result, an FRP term does not always have a meaningful semantics. In such cases we say that the term denotes \perp (and is thus denotationally equivalent to non-termination or error). Of course, this is not very informative. Even worse, it is possible to write egregious behaviors for which either the implementation does not converge when we increase the sampling rate, or it converges to something other than its semantics. However, we are able to identify a set of sufficient conditions which guarantees the fidelity of the implementation. These conditions are neither complete nor decidable in general, which means the burden of “good behavior” is on the programmer. However, it is perhaps not surprising given such a rich mathematical language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI 2000, Vancouver, British Columbia, Canada.
Copyright 2000 ACM 1-58113-199-2/00/0006...\$5.00.

2. AN INTRODUCTION TO FRP

In this section we give a very brief introduction to FRP; see [5, 7] for more details. FRP is an example of an *embedded domain-specific language* [9]. In our case the “host” is Haskell [11], a higher-order, typed, polymorphic, lazy and purely functional language, and thus all of our examples (as well as our implementation) are in Haskell syntax.

There are two key polymorphic data types in FRP: the *Behavior* and the *Event*. A value of type *Behavior* *a* is a value of type *a* that varies over continuous time. Constant behaviors include numbers (such as `1 :: Behavior Real`), colors (such as `red :: Behavior Color`), and others. The most basic time-varying behavior is time itself: `time :: Behavior Time`, where *Time* is a synonym for *Real*. More interesting time-varying behaviors include animations of type *Behavior Picture* (which is the key idea behind Fran [5, 7], a language for functional reactive animations), sonar readings of type *Behavior Sonar*, velocity vectors of type *Behavior (Real,Real)*, and so on (the latter two examples are used in Frob [13, 14], an FRP-based language for controlling robots). (Note: In our implementation the type *Real* is approximated by *Float*.)

A value of type *Event* *a* is a time-ordered sequence of event occurrences, each carrying a value of type *a*. Basic events include left button presses and keyboard presses, represented by the values `lbp :: Event ()` and `key :: Event Char`, respectively. The declarative reading of `lbp` (and `key`) is that it is an event *sequence* containing all of the left button presses (and key presses), not just one.

Behaviors and events are both first-class values in FRP, and there is a rich set of operators (combinators) that the user can use to compose new behaviors and events from existing ones. An FRP *program* is just a set of mutually-recursive behaviors and events, each of them built up from static (non-time-varying) values and/or other behaviors and events.

Suppose that we wish to generate a color behavior which starts out as red, and changes to blue when the left mouse button is pressed. In FRP we would write:

```
> color :: Behavior Color
> color = red 'until' (lbp ==> blue)
```

This can be read “behave as red until the left button is pressed, then change to blue.” We can then use `color` to color an animation, as follows:

```
> ball :: Behavior Picture
> ball = paint color circ
>
> circ :: Behavior Region
> circ = translate (cos time, sin time) (circle 1)
```

Here `circle 1` creates a circle with radius 1, and the translation causes it to revolve about the center of the screen with period 2π seconds. Thus `ball` is a revolving circle that changes from red to blue when the left mouse button is pressed.

Sometimes it is desirable to choose between two different behaviors based on user input. For example, this version of `color`:

```
> color2 = red 'until'
>         (lbp ==> blue) .|. (key ==> yellow)
```

will start off as red and change to blue if the left mouse button is pressed, or to yellow if a key is pressed. The `.|.` operator can be read as the “or” of its event arguments.

The function `when` transforms a Boolean behavior into an event that occurs exactly “when” the Boolean behavior becomes *True*; this is called a *predicate event*. For example:

```
> color3 = red 'until'
>         (when (time > 5) ==> blue)
```

defines a color that starts off as red and becomes blue after time is greater than 5.

Sometimes it is desirable to “lift” an ordinary value or function to an analogous behavior. The family of functions

```
> lift0 :: a -> Behavior a
> lift1 :: (a -> b) -> (Behavior a -> Behavior b)
```

and so on, perform such coercions in FRP. Sometimes Haskell overloading permits us to use the same name for lifted and unlifted functions, such as most of the arithmetic operators. When this is not possible, we use the convention of placing a “*” after the unlifted function name. For example, `>*` in the `color3` example is the lifted version of `>`.

Finally, one of the most useful operations in FRP is *integration* of numeric behaviors over time. For example, the physical equations that describe the position of a mass under the influence of an accelerating force *f* can be written as:

```
> s,v :: Behavior Real
> s = s0 + integral v
> v = v0 + integral f
```

where *s0* and *v0* are the initial position and velocity, respectively. Note the similarity of these equations to the mathematical equations describing the same physical system:

$$\begin{aligned}s(t) &= s_0 + \int_0^t v(\tau) d\tau \\ v(t) &= v_0 + \int_0^t f(\tau) d\tau\end{aligned}$$

This example demonstrates well the declarative nature of FRP. A major design goal for FRP is to free the programmer from “presentation” details by providing the ability to think in terms of “modeling.” It is common that an FRP program is concise enough to also serve as a specification for the problem it solves.

There are many other useful operations in FRP, but we introduce them only as needed in the remainder of the paper.

3. THE SEMANTIC FRAMEWORK

In this section we present the semantic framework for behaviors and events. The semantics of each FRP construct will be given individually in Section 6.

FRP's notion of continuous time is denoted by the domain *Time*, which is a synonym for the set of real numbers \mathbb{R} . Let $\langle \text{Behavior}_\alpha \rangle$ and $\langle \text{Event}_\alpha \rangle$ denote the set of all FRP terms of type *Behavior* α and *Event* α respectively, where α is any Haskell data type. The meaning of behaviors and events is given by the following semantic functions:

$$\begin{aligned} \text{at} & : \langle \text{Behavior}_\alpha \rangle \rightarrow \text{Time} \rightarrow \text{Time} \rightarrow \alpha \\ \text{occ} & : \langle \text{Event}_\alpha \rangle \rightarrow \text{Time} \rightarrow \text{Time} \rightarrow [\text{Time} \times \alpha] \end{aligned}$$

where $[-]$ is the list type constructor.

Intuitively, the meaning of a behavior, as given by **at**, is a function mapping a start time and a time of interest to the value of the behavior at the given time of interest. Start times relate to the reactive nature of FRP. For example, in $(b \text{ 'until' } e)$, if an event occurrence (t, b') of e causes the overall behavior to switch to b' , we say that b' starts at time t . A behavior is unaware of any event occurrences that happened before its start time.

The meaning of an event, given by **occ**, is a function that takes also a start time T and a time of interest t , and returns a finite list of time-ascending occurrences of the event in the interval $(T, t]$. The start time of an event is analogous to the start time of a behavior. Note that the lower end of the interval is open, which means an occurrence precisely at the start time is not detected.

Note that, for simplicity, we have omitted real-world events such as user input and general I/O from this semantic framework. However, predicate events such as described in the last section are still present, which are sufficient to demonstrate all interesting aspects of the semantics and implementation. Nevertheless, for completeness, we describe how to add user input in Section 8.

4. A STREAM IMPLEMENTATION OF FRP

Our stream-based implementation of FRP is interesting in its own right, but because of space limitations we omit a detailed discussion of it here; the basic idea is outlined in [6] and elaborated in [10].

The core data types in FRP, *Behavior* and *Event*, are given by:

```
> type Behavior a = [Time] -> [a]
> type Event a    = [Time] -> [Maybe a]
```

Here *Maybe a* is a data type whose values are either *Nothing* or *Just x*, where x is some a .

Intuitively, a behavior is a stream transformer: a function that takes an infinite stream of sample times, and yields an infinite stream of values representing its behavior. Similarly, an event is also a stream transformer, and can be thought of as a behavior where, at each time t , the event either

occurs (indicated as *Just x* for some x), or does not occur (indicated as *Nothing*). Note that using this implementation strategy for events means that we must ensure that the time associated with each event occurrence actually appears in the time stream, but this is easily done.

The implementation itself can be divided into two parts: (1) definitions of FRP's primitive behaviors, events, and combinators as stream transformers, and (2) a "run-time system" that interprets the behaviors and events by building an infinite stream of sample times and applying the behavior/event to the stream. The latter task is explained in the remainder of this section; we return to the former in Section 6.

To simplify the presentation of the run-time system, we omit the interface to the operating system that extracts events, grabs the clock time, etc. The resulting abstract implementation is captured by the following pair of "interpreters," one for behaviors, the other for events:

$$\begin{aligned} \tilde{\text{at}} & : \langle \text{Behavior}_\alpha \rangle \rightarrow [\text{Time}] \rightarrow \alpha \\ \tilde{\text{at}}[b] \text{ } ts & \stackrel{\text{def}}{=} \text{last } ([b] \text{ } ts) \\ \tilde{\text{occ}} & : \langle \text{Event}_\alpha \rangle \rightarrow [\text{Time}] \rightarrow [\text{Time} \times \alpha] \\ \tilde{\text{occ}}[e] \text{ } ts & \stackrel{\text{def}}{=} \text{justValues } ts \text{ } ([e] \text{ } ts) \end{aligned}$$

where (1) we write $[-]$ for the value (which could be a function) denoted by the Haskell term $-$, (2) *last* returns the last element of a list, and (3) the auxiliary function:

$$\text{justValues} : [\text{Time}] \rightarrow [\text{Maybe } \alpha] \rightarrow [\text{Time} \times \alpha]$$

time-stamps a stream of "Maybe" values while dropping the "Nothing's." For example,

```
justValues [0.0, 0.1, 0.2, 0.3, 0.4]
[Nothing, Just False, Nothing, Nothing, Just True]
```

returns $[(0.1, \text{False}), (0.4, \text{True})]$.

Intuitively, $\tilde{\text{at}}$ takes a behavior and an ordered finite list of sample *Time*'s, the first in the list being the start time of the behavior and the last being the time of interest. It returns as result the value of the behavior at the time of interest. Similar is $\tilde{\text{occ}}$, which returns all the occurrences detected up until the time of interest. In essence, $\tilde{\text{at}}$ and $\tilde{\text{occ}}$ define an operational semantics for FRP. (Note that b is a Haskell term of type $[\text{Time}] \rightarrow [\alpha]$, so $[b]$ is a function of type $[\text{Time}] \rightarrow [\alpha]$, and therefore $[b] \text{ } ts$ is a value of type $[\alpha]$.)

In this paper we are most interested in the limit of the operational semantics as the sampling interval goes to zero. Thus we define:

$$\begin{aligned} \tilde{\text{at}}^* & : \langle \text{Behavior}_\alpha \rangle \rightarrow \text{Time} \rightarrow \text{Time} \rightarrow \alpha \\ \tilde{\text{at}}^*[b] \text{ } T \text{ } t & \stackrel{\text{def}}{=} \begin{cases} \lim_{|P_T^t| \rightarrow 0} \tilde{\text{at}}[b] P_T^t & \text{such limit exists} \\ \perp & \text{otherwise} \end{cases} \\ \tilde{\text{occ}}^* & : \langle \text{Event}_\alpha \rangle \rightarrow \text{Time} \rightarrow \text{Time} \rightarrow [\text{Time} \times \alpha] \\ \tilde{\text{occ}}^*[e] \text{ } T \text{ } t & \stackrel{\text{def}}{=} \begin{cases} \lim_{|P_T^t| \rightarrow 0} \tilde{\text{occ}}[e] P_T^t & \text{such limit exists} \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

where P_T^t is a partition of $[T, t]$.

Definition 1. (Partition and norm of partition) A *partition* P of a closed interval $[a, b]$ is a non-empty finite list $[x_0, x_1, \dots, x_n]$, such that $a = x_0 < x_1 < \dots < x_n = b$, where $n \geq 0$. Such a partition is often written as P_a^b . The *norm* of P , written as $|P|$, is defined as the maximum of the set $\{x_i - x_{i-1} \mid 1 \leq i \leq n\}$ when $n \geq 1$, or 0 when $n = 0$.

(Note that we overload the notation $[a, b]$ for both a closed interval and a list of two elements, and similarly (a, b) for both an open interval and a tuple. However, the meaning is always clear from context.)

5. FAITHFUL IMPLEMENTATIONS AND UNIFORM CONVERGENCE

In the next section we will give both the denotational semantics and stream-based implementation of each FRP construct in turn, and show in each case that the implementation is *faithful* to the semantics, though possibly only under certain constraints, in the following formal sense:

$$\begin{aligned} \tilde{at}^*[b] T t &= at[b] T t \\ \tilde{occ}^*[e] T t &= occ[e] T t \end{aligned} \quad (1)$$

In addition, we will identify the cases where the implementation *converges uniformly*, a property (defined below) that is necessary to ensure that the integral of a numeric behavior is well defined. Analogous to the concept of uniform convergence for real number function series [1, page 393], we define *uniform convergence* for functions defined on partitions of real intervals:

Definition 2. (Uniform Convergence) Given a set S , we say that a function F defined on \mathcal{P}_T , which is the set of all partitions whose left (i.e. smaller) end is T , *converges uniformly* to f on S if, for every $\epsilon > 0$, there exists a $\delta > 0$ (depending only on ϵ) such that for every $t \in S$ and P_T^t satisfying $|P_T^t| < \delta$,

$$|F(P_T^t) - f(t)| < \epsilon$$

We denote this symbolically by writing

$$F(P_T^t) \mapsto f(t) \text{ uniformly on } S$$

So, when possible, we will indicate when the following condition holds:

$$\tilde{at}[b] P_T^t \mapsto at[b] T t \text{ uniformly} \quad (2)$$

Note that (2) implies (1), and for conciseness we will not write out (1) if we have already established (2).

Because FRP is an “embedded” DSL, it is difficult to draw a clear line between FRP and Haskell. Thus a full treatment of the FRP “language” inevitably requires a treatment of all of Haskell. As this would obscure our main interest, we choose to discuss only the constructs specific to FRP.

6. CORRESPONDANCE BETWEEN SEMANTICS AND IMPLEMENTATION

The proof of the important theorems in this section can be found in appendix A.

Time

The primitive behavior time is implemented as:

```
> time :: Behavior Time
> time = \ts -> ts
```

and its semantics is given by:

$$at[time] T t = t$$

We can show that the implementation of time is faithful to its semantics, and that its convergence is uniform:

THEOREM 1. $\tilde{at}[time] P_T^t \mapsto t$ uniformly on $(-\infty, \infty)$.

Lifting

Here we show the correctness of the three most useful lifting operators: `lift0`, `lift1` and `lift2`. The result easily extends to any arity of lifting. The lifting operators are implemented as:

```
> ($) :: Behavior (a -> b)
>      -> Behavior a -> Behavior b
> ff $* fb =
>   \ts -> zipWith ($) (ff ts) (fb ts)
>
> lift0 :: a -> Behavior a
> lift0 x = map (const x)
>
> lift1 :: (a -> b) -> (Behavior a -> Behavior b)
> lift1 f b1 = lift0 f $* b1
>
> lift2 :: (a -> b -> c)
>        -> (Behavior a
>            -> Behavior b -> Behavior c)
> lift2 f b1 b2 = lift1 f b1 $* b2
```

The semantics of `lift0` is given by:

$$at[lift0 c] T t = [c]$$

Unsurprisingly, the implementation converges to the semantics uniformly:

THEOREM 2. $\tilde{at}[lift0 c] P_T^t \mapsto [c]$ uniformly on $(-\infty, \infty)$.

The semantics of `lift1` is given by:

$$at[lift1 f b] T t = [f] (at[b] T t)$$

The implementation of `lift1` is faithful to its semantics, but only when the lifted function is continuous:

THEOREM 3. If $\tilde{at}^*[b] T t = b_t$, and $[f]$ is continuous at b_t , then $\tilde{at}^*[lift1 f b] T t = [f] b_t$.

It is worth noting that we only require $[f]$ to be continuous at b_t , not necessarily continuous everywhere. Since most

functions we deal with in FRP are either globally continuous or piecewise continuous, the theorem applies in most cases.

To see whether the convergence of `lift1` is uniform, we need a concept called *uniform continuity* [1, page 74], as found in most treatments of calculus:

Definition 3. (Uniform Continuity) A function f is said to be *uniformly continuous* on a set S if for every $\epsilon > 0$, there exists a $\delta > 0$ (depending only on ϵ) such that if $x, y \in S$ and $|x - y| < \delta$, then $|f(x) - f(y)| < \epsilon$.

THEOREM 4. If $\tilde{at}[b] P_T^t \mapsto fb(t)$ uniformly on S , and $[f]$ is uniformly continuous, then

$$\tilde{at}[\text{lift1 } f \ b] P_T^t \mapsto [f] (fb(t)) \text{ uniformly on } S$$

For example, the lifted sin function is defined as:¹

```
> instance Floating a => Floating (Behavior a) where
> sin = lift1 sin
```

Note that the `sin` on the right hand side of the definition is the static version as in the standard Haskell library:

```
> sin :: Floating a => a -> a
```

As an example, we can use theorem 4 to prove that the expression “`sin time`” in FRP actually denotes the mathematical notion of $\sin(t)$, where t is the current time.

COROLLARY 1. $\tilde{at}[\text{sin time}] P_T^t \mapsto \sin t$ uniformly on $(-\infty, \infty)$.

The semantics of `lift2` is given by:

$$\text{at}[\text{lift2 } f \ b \ d] T \ t = [f] (\text{at}[b] T \ t) (\text{at}[d] T \ t)$$

Similar to `lift1`, we can show:

THEOREM 5. If $\tilde{at}[b] T \ t = b_t$, $\tilde{at}[d] T \ t = d_t$, and $(\text{uncurry } [f])$ is continuous at (b_t, d_t) , then

$$\tilde{at}[\text{lift2 } f \ b \ d] T \ t = [f] \ b_t \ d_t.$$

as well as the following for uniform convergence:

THEOREM 6. If $\tilde{at}[b] P_T^t \mapsto fb(t)$ uniformly on S , $\tilde{at}[d] P_T^t \mapsto fd(t)$ uniformly on S , and $(\text{uncurry } [f])$ is uniformly continuous, then

$$\tilde{at}[\text{lift2 } f \ b \ d] P_T^t \mapsto [f] (fb(t)) (fd(t))$$

uniformly on S .

¹The instance declaration shown here is how, using Haskell’s type class system, functions are overloaded. In this case, `sin` is a method in the class `Floating`, and the instance declaration says that `sin` may now be used for values of type `Behavior a`, for any type `a` that is already an instance of the class `Floating`.

For example, we can use this theorem to verify the semantics of the lifted binary operator `+`:

```
> instance Num a => Num (Behavior a) where
> (+) = lift2 (+)
```

COROLLARY 2. $\tilde{at}^*[b + d] T \ t = \tilde{at}^*[b] T \ t + \tilde{at}^*[d] T \ t$

Integration

We use a very simple numerical algorithm to calculate the Riemann integration of numeric behaviors:

```
> integral :: Behavior Real -> Behavior Real
> integral fb =
>   \ts@(\t:ts') -> 0 : loop t 0 ts' (fb ts)
>   where loop t0 acc (t1:ts) (a:as)
>         = let acc' = acc + (t1-t0)*a
>           in acc' : loop t1 acc' evs ts as
```

The formal semantics of `integral` is given by:

$$\text{at}[\text{integral } f] T \ t = \int_T^t (\text{at}[f] T \ \tau) d\tau$$

As mentioned earlier, this stream-based integrator is only sound mathematically if the behavior to be integrated converges uniformly:

THEOREM 7. If $\tilde{at}[b] P_T^t \mapsto fb(\tau)$ uniformly on $[T, t]$, then

$$\tilde{at}[\text{integral } b] P_T^t \mapsto \int_T^t fb(\eta) d\eta$$

uniformly on $[T, t]$.

If $\tilde{at}[b] P_T^t \mapsto fb(\tau)$ non-uniformly, we can say nothing about $\tilde{at}^*[\text{integral } b] T \ t$. As an instance, consider the behavior `bizarre` (inspired by [1, page 401]), which is non-uniformly convergent on $[0, 1]$, and is defined as:

```
> bizarre :: Behavior Real
> bizarre = 0 'until' e ==> b
>   where e = when (time >= 0) 'snapshot' time
>         b = \(_,t1) -> c*c*time*(1 - time)**c
>           where c = 1/t1
```

On time interval $[0, 1]$, the above is equivalent to:

```
> bizarre = \(\t0:t1:ts) ->
>   0 : 0 : map (let c = 1/(t1 - t0)
>                 in \t -> c*c*t*(1 - t)**c) ts
```

When $t \in [0, 1]$, we have

$$\begin{aligned} & \tilde{at}^*[\text{bizarre}] 0 \ t \\ &= \lim_{|P_0^t| \rightarrow 0} \text{last } ([\text{bizarre}] P_0^t) \\ &= \lim_{|P_0^t| \rightarrow 0} c^2 t (1 - t)^c, \text{ where} \\ & \quad 1/c = \text{the length of the first sub-interval of } P_0^t \\ &= 0 \end{aligned}$$

Hence $\int_0^1 (\tilde{at}^*[\text{bizarre}] 0 t) dt = 0$. However, we have

$$\begin{aligned} & \tilde{at}^*[\text{integral bizarre}] 0 1 \\ &= \lim_{|P_0^1| \rightarrow 0} \sum_{i=1}^n ([\text{bizarre}] P_0^1)_{(i)} \cdot \Delta t_i \\ & \quad \text{where } P_0^1 = [t_0 = 0, t_1, \dots, t_n = 1], \text{ and} \\ & \quad \text{subscript } (i) \text{ means the } i\text{-th element of a list} \\ &= \lim_{|P_0^1| \rightarrow 0} \sum_{i=2}^n c^2 t_{i-1} (1 - t_{i-1})^c \cdot \Delta t_i, \\ & \quad \text{where } c = 1/\Delta t_1 \end{aligned}$$

and

$$\lim_{c \rightarrow \infty} \int_0^1 c^2 t (1 - t)^c dt = \lim_{c \rightarrow \infty} \frac{c^2}{(c+1)(c+2)} = 1$$

Therefore

$$\begin{aligned} & \tilde{at}^*[\text{integral bizarre}] 0 1 \\ & \neq \int_0^1 (\tilde{at}^*[\text{bizarre}] 0 t) dt \end{aligned}$$

In other words, the limit of the integral doesn't agree with the integral of the limit for **bizzare**.

It turns out that global uniform convergence is usually too strong a condition to achieve in practice, part of the reason being that the interplay of behaviors and events often results in \perp at the point of behavior switching. The following theorem relaxes the requirement considerably:

THEOREM 8. *If $\tilde{at}[b] P_T^\tau \mapsto fb(\tau)$ uniformly on $[T, t]$, except at n (finite) points $\tau_1, \tau_2, \dots, \tau_n$, and $\tilde{at}[b] P_T^{\tau_i}$ is bounded as $|P_T^{\tau_i}| \rightarrow 0$ for every $1 \leq i \leq n$, then*

$$\tilde{at}[\text{integral } b] P_T^\tau \mapsto \int_T^\tau F(\eta) d\eta$$

uniformly on $[T, t]$, where

$$F(\eta) = \begin{cases} fb(\eta) & \eta \neq \tau_i \text{ for every } 1 \leq i \leq n \\ \text{any finite value} & \text{otherwise} \end{cases}$$

Since most behaviors we encounter in practice are bounded on every finite interval, the above condition is not hard to satisfy.

As shown in theorem 7, integration preserves the uniform convergence property. This allows us to safely calculate the second integral, the third, and so on:

COROLLARY 3. *If $\tilde{at}[b] P_T^\tau \mapsto fb(\tau)$ uniformly on $[T, t]$, then*

$$\begin{aligned} & \tilde{at}[\text{integral } (\text{integral } \dots (\text{integral } b) \dots)] P_T^\tau \\ & \mapsto \int_T^\tau \int_T^{\tau_1} \dots \int_T^{\tau_{n-1}} fb(\tau_n) d\tau_n d\tau_{n-1} \dots d\tau_1 \end{aligned}$$

uniformly on $[T, t]$.

Event Mapping

The \Rightarrow operator essentially maps a function over the event stream, and is implemented as:

```
> (==>) :: Event a -> (a->b) -> Event b
> fe ==> f = map (map f) . fe
```

Its semantics is given by:

$$\begin{aligned} \text{occ}[e \Rightarrow f] T t &= \\ & [(t_1, [f] v_1), (t_2, [f] v_2), \dots, (t_n, [f] v_n)], \text{ where} \\ \text{occ}[e] T t &= [(t_1, v_1), (t_2, v_2), \dots, (t_n, v_n)] \end{aligned}$$

THEOREM 9. *If $\tilde{occ}^*[e] T t = [(t_1, v_1), (t_2, v_2), \dots, (t_n, v_n)]$, and $[f]$ is continuous at v_i for every $1 \leq i \leq n$, then*

$$\begin{aligned} \tilde{occ}^*[e \Rightarrow f] T t &= \\ & [(t_1, [f] v_1), (t_2, [f] v_2), \dots, (t_n, [f] v_n)] \end{aligned}$$

The \Rightarrow operator we used previously is just syntactic sugar:

$$e \Rightarrow b \stackrel{\text{def}}{=} e \Rightarrow \backslash _ \rightarrow b$$

Choice

\mid can be used to merge two events of the same type; it is implemented as:

```
> (.|.) :: Event a -> Event a -> Event a
> fe1 .|. fe2 =
> \ts -> zipWith aux (fe1 ts) (fe2 ts)
>   where aux Nothing Nothing = Nothing
>         aux (Just x) _       = Just x
>         aux _ (Just x)       = Just x
```

The semantics of \mid operator is given by:

If $\text{occ}[e_1] T t = [(t_1, v_1), (t_2, v_2), \dots, (t_n, v_n)]$, $\text{occ}[e_2] T t = [(t'_1, v'_1), (t'_2, v'_2), \dots, (t'_m, v'_m)]$, and $t_1, t_2, \dots, t_n, t'_1, t'_2, \dots, t'_m$ are distinct, then

$$\text{occ}[e_1 \mid e_2] T t = [(\tau_1, w_1), (\tau_2, w_2), \dots, (\tau_{n+m}, w_{n+m})],$$

where $ts = \{t_1, t_2, \dots, t_n, t'_1, t'_2, \dots, t'_m\}$, and

$$(\tau_i, w_i) = \begin{cases} (t_j, v_j) & t_j \text{ is the } i\text{-th smallest of } ts \\ (t'_k, v'_k) & t'_k \text{ is the } i\text{-th smallest of } ts \end{cases}$$

Note that we require the occurrence times are distinct, because the result of merging two simultaneous occurrences is nondeterministic.

We can show that the implementation of \mid converges to its semantics. To save space, we don't give the formal statement here.

Behavior Switching

The until operator is implemented as:

```
> until :: Behavior a -> Event (Behavior a)
>       -> Behavior a
> fb 'until' fe =
> \ts -> loop ts (fe ts) (fb ts)
>   where loop ts@(_:ts') ~(e:es) (b:bs) =
>         b : case e of
>           Nothing -> loop ts' es bs
>           Just fb' -> tail (fb' ts)
```

and its semantics is given by:

If $\text{occ}[e] T t = [(t_1, [b_1]), \dots, (t_n, [b_n])]$, then for any $\tau \in [T, t]$,

$$\text{at}[b \text{ 'until' } e] T \tau = \begin{cases} \text{at}[b] T \tau & n = 0 \text{ or } \tau \leq t_1 \\ \text{at}[b_1] t_1 \tau & \text{otherwise} \end{cases}$$

This operator is precisely where behaviors interact with events, and thus its good behavior is critical to the goodness of FRP. We can show:

THEOREM 10. If $\widetilde{\text{occ}}^*[e] T t = [(t_1, [b_1]), \dots, (t_n, [b_n])]$, then for any $\tau \in [T, t]$ where $\tau \neq t_1$,

$$\widetilde{\text{at}}^*[b \text{ 'until' } e] T \tau = \begin{cases} \widetilde{\text{at}}^*[b] T \tau & n = 0 \text{ or } \tau < t_1 \\ \lim_{\eta \rightarrow t_1} \widetilde{\text{at}}^*[b_1] \eta \tau & \text{otherwise} \end{cases}$$

Most behaviors are continuous with respect to their start times (i.e. a small change in the start time only results in a small change in the result value); for example this is true of *integral*. Some behaviors are even independent of the start time, as with *time*. In such cases, the limit operator in the above theorem can be dropped, and the implementation becomes consistent with the semantics.

Snapshot

snapshot samples a behavior at the exact moments an event occurs.

```
> snapshot :: Event a -> Behavior b
>          -> Event (a,b)
> snapshot fe fb
> = \ts -> zipWith aux (fe ts) (fb ts)
>   where aux (Just x) y = Just (x,y)
>   aux Nothing _ = Nothing
```

The semantics:

If $\text{occ}[e] T t = [(t_1, a_1), (t_2, a_2), \dots, (t_n, a_n)]$ and $\text{at}[b] T t_i = b_i$, then

$$\text{occ}[e \text{ 'snapshot' } b] T t = [(t_1, (a_1, b_1)), (t_2, (a_2, b_2)), \dots, (t_n, (a_n, b_n))]$$

The implementation is faithful to the semantics unconditionally:

THEOREM 11. If $\widetilde{\text{occ}}^*[e] T t = [(t_1, a_1), (t_2, a_2), \dots, (t_n, a_n)]$ and $\widetilde{\text{at}}^*[b] T t_i = b_i$, then

$$\widetilde{\text{occ}}^*[e \text{ 'snapshot' } b] T t = [(t_1, (a_1, b_1)), (t_2, (a_2, b_2)), \dots, (t_n, (a_n, b_n))]$$

Predicate Events

We can turn a Boolean behavior into an event that occurs every time the behavior changes from *False* to *True*. To do so we use the *when* combinator defined as:

```
> when :: Behavior Bool -> Event ()
> when fb =
>   \ts -> zipWith up (True : bs) bs
>   where bs = fb ts
>   up False True = Just ()
>   up _ _ = Nothing
```

We define the semantics of *when* as follows: Given $T, t \in \text{Time}$, let $fb(\tau) = \text{at}[b] T \tau$. If there are $c_1, c_2, \dots, c_n \in \text{Bool}$ and a partition $[t_0, t_1, \dots, t_n]$ of $[T, t]$, such that:

1. For all $1 \leq i \leq n-1$, $c_i = \neg c_{i+1}$;
2. For all $1 \leq i \leq n-1$, $\tau \in (t_{i-1}, t_i)$ implies $fb(\tau) = c_i$;
3. $fb(T) \neq \perp$ or $c_1 = \text{False}$, and
4. $fb(t) \neq \perp$ or $c_n = \text{True}$.

then $\text{occ}[\text{when } b] T t = \text{occs} \neq \perp$, where *occs* is the shortest time-ascending list satisfying:

1. If $c_1 = \text{True}$ and $fb(T) = \text{False}$, then $(T, ()) \in \text{occs}$;
2. For $1 \leq i \leq n-1$, $(t_i, ()) \in \text{occs}$ if $c_i = \text{False}$, and
3. If $c_n = \text{False}$ and $fb(t) = \text{True}$, then $(t, ()) \in \text{occs}$.

Otherwise $\text{occ}[\text{when } b] T t = \perp$.

This may seem complicated, but it basically says that *when* *b* has an occurrence at time τ iff $\text{at}[b] T \eta$ (viewed as a function of η) jumps from *False* to *True* as η crosses point τ from the left (i.e. the negative side).

According to this rule, if $fb(\tau)$ toggles its value back and forth instantaneously at some $\tau_0 \in (T, t)$, then $\text{occ}[\text{when } b] T t = \perp$. To see why this is true, suppose $\text{occ}[\text{when } b] T t \neq \perp$, then there must be c_1, \dots, c_n and t_0, \dots, t_n satisfying the above constraints. In addition, τ_0 must be equal to t_k for some $1 \leq k \leq n-1$, because $fb(\tau)$ remains constant in each of (t_{i-1}, t_i) where $1 \leq i \leq n$. However, this means $c_k = c_{k+1}$, which violates the constraint $c_i = \neg c_{i+1}$.

This rule implies that on any finite time interval, a predicate event can only occur a finite number of times (for the number *n* above is finite). Therefore, if the Boolean behavior ever oscillates at an infinite frequency (we will see such an example later), the semantics of the event is \perp .

The implementation of *when* *b* is faithful to the semantics if the implementation of *b* converges uniformly:

THEOREM 12. Given a time interval $[T, t]$, if $\widetilde{\text{at}}[b] P_T^r \mapsto fb(\tau)$ uniformly on $[T, t]$, then

$$\widetilde{\text{occ}}^*[\text{when } b] T t = \text{occ}[\text{when } b] T t.$$

We require $\widetilde{\text{at}}[b] P_T^r \mapsto fb(\tau)$ uniformly on $[T, t]$, because the mere existence of $\widetilde{\text{at}}^*[b] T \tau$ is not sufficient here. For

example, given the behavior `bizarre` we discussed in Section 6, $\text{at}^*[\text{bizarre} > 1] 0 \tau = \text{False}$ for every $\tau \in [0, 1]$, and thus $\text{occ}[\text{when} (\text{bizarre} > 1)] 0 1 = []$, but

$$\widetilde{\text{occ}}^*[\text{when} (\text{bizarre} > 1)] 0 1 = [(0, ())] \neq []$$

7. EGREGIOUS BEHAVIORS AND EVENTS

As mentioned earlier, it is possible to define certain egregious behaviors and events in FRP, and a good understanding of them is helpful in understanding the semantic rules and theorems that we have introduced. Consider first this event:

```
> sharp :: Event ()
> sharp = when (time == 1)
```

This looks innocent enough, but the predicate is true only instantaneously at time = 1. To sample `sharp`, let's consider a series of partitions $\{P_n \mid n \in \mathbb{N}\}$ of $[0, 2]$, where $P_n = [0, \frac{1}{2n+1}, \frac{2}{2n+1}, \dots, \frac{4n+1}{2n+1}, 2]$. Obviously $\lim_{n \rightarrow \infty} |P_n| = 0$, however none of the partitions divides $[0, 2]$ at point 1. Hence our sampling based implementation could fail to find the event occurrence at time 1. This explains why our semantic rule for `when` gives \perp as the denotation of `sharp`.

It is worth pointing out that one can write a well-behaved definition for `sharp`:

```
> sharp2 = when (time >= 1)
```

Consider next this encoding of *Zeno's Paradox*:

```
> zeno :: Event ()
> zeno = when (lift1 f time) where
>   f t = t < 2 && even (floor (log2 (2 - t)))
```

This example demonstrates an infinitely dense sequence of events at times $t_0 = 1$, $t_1 = 1.5$, $t_2 = 1.75$, and in general $t_{n+1} = t_n + 2^{-(n+1)}$. This creates obvious problems with an implementation. But even if we could implement such event sequences, there is a more fundamental semantic problem. Suppose there is an electric light in a room, initially off. A daemon comes in and turns the light on at time t_0 , off at time t_1 , and so on. A simple calculation shows that $\lim_{n \rightarrow \infty} t_n = 2$, so the whole process comes to an end at time $t = 2$. Now the question is: is the light on or off after the daemon stops? The answer might be surprising: it could be either on or off; i.e., this is a natural expression of nondeterminism. Our semantic rules give that $\text{occ}[\text{zeno}] T t = \perp$ for any $T < 2 \leq t$, which provides no information about what the implementation will give us, and therefore is compatible with the nondeterministic semantics one might expect.

Finally, consider this unpredictable behavior:

```
> unpredictable :: Behavior Real
> unpredictable =
>   0 'until' (when (time > 1)
>     'snapshot' sin (1/(time - 1)))
>   ==> (\(_, x) -> lift0 x)
```

In a stream-based implementation, we don't know what value `unpredictable` will yield at run time, since it depends on the sampling frequency and phase. For this example, the semantic rules give a value in terms of $\sin \frac{1}{t-1}$ where $t = 1$, which is undefined due to the discontinuity of the function.

`sharp`, `zeno` and `unpredictable` are all examples where the semantics is \perp . There are other egregious values where the semantics is not \perp , but the implementation does not agree with it. `integral bizarre` is one of them.

8. INTERFACE WITH REAL WORLD

As was mentioned earlier, behaviors and events can also react to user actions, which we can capture formally through the notion of an *environment*, which can be viewed as a finite set of primitive behaviors and events. Thus, strictly speaking, the semantic functions should have type:

```
at   : (Behavior $_{\alpha}$ )  $\rightarrow$  Env  $\rightarrow$  Time  $\rightarrow$  Time  $\rightarrow$   $\alpha$ 
occ  : (Event $_{\alpha}$ )  $\rightarrow$  Env  $\rightarrow$  Time  $\rightarrow$  Time  $\rightarrow$  [Time  $\times$   $\alpha$ ]
```

where *Env* is the abstract data type for all the input to the FRP system.

For example, in *Fran*, the environment includes mouse movements, mouse button presses, and keyboard presses. Thus we can define $\text{Env} = \text{PBeh}_{\text{Int} \times \text{Int}} \times \text{PEvt}_{()} \times \text{PEvt}_{()} \times \text{PEvt}_{\text{char}}$, where the four components of the tuple correspond to mouse position, left button press, right button press and keyboard press, respectively. Type of the form PBeh_{α} and PEvt_{α} are defined as:

$$\begin{aligned} \text{PBeh}_{\alpha} &= \text{Time} \rightarrow \alpha \\ \text{PEvt}_{\alpha} &= \text{Time} \rightarrow [\text{Time} \times \alpha] \end{aligned}$$

This idea can be justified by noting that primitive behaviors and events are in fact observations of physical signals outside of the FRP system. Their values at a particular time do not depend on when we start the observation. Therefore a value of type PBeh_{α} is just a mapping from the current time to the value, and a value of type PEvt_{α} is a mapping from current time to all the occurrences since the initialization of the system.

The treatment of environment is almost orthogonal to the treatment of the individual FRP operators. This allows us to address the issue separately. To extend the stripped-down version of FRP to incorporate environments, we need only to:

1. Define the semantics for the FRP constructs that represent user actions.

For `mouse :: Behavior (Int, Int)`, we have:

$$\text{at}[\text{mouse}] (\text{mouse}, \text{lbp}, \text{rbp}, \text{key}) T t = \text{mouse } t.$$

For `lbp :: Event ()`, we have:

$$\text{occ}[\text{lbp}] (\text{mouse}, \text{lbp}, \text{rbp}, \text{key}) T t = \text{after } T (\text{lbp } t),$$

where *after* *T list* drops all elements in *list* whose time-stamp is less than or equal to *T*.

2. Pass the environment parameter around in the semantic equations for composite behaviors/events. For in-

stance, the meaning for `lift2` is now given by:

$$\text{at}[\text{lift2 } f \ b_1 \ b_2] \ \text{env } T \ t = \\ [f] \ (\text{at}[b_1] \ \text{env } T \ t) \ (\text{at}[b_2] \ \text{env } T \ t)$$

9. CONCLUSIONS AND RELATED WORK

Although the signal processing literature is full of foundational work on the validity and accuracy of sampling techniques, we are not aware of any work attempting to define the semantics of a reactive programming language such as FRP. Also, most of the signal processing work shies away from discontinuous signals, whereas we have shown that under the right conditions values are still well behaved.

In [7] we described a denotational semantics for Fran. The semantics given in this paper is different in that it parameterizes the start time for behaviors and events, and contains a more precise characterization of events. Various implementation techniques for Fran are discussed in [6], including the basic ideas behind a stream-based implementation; in [10] the particular implementation used in this paper is described in detail.

It is worth noting that we concentrated here on just one implementation technique for FRP; it may well be that other techniques either have more or fewer constraints than those discovered for streams. In particular, it is worth pointing out that *interval analysis* can be used to safely capture instantaneous predicate events [6, 7].

Finally, we point out that all of our results depend on sufficient accuracy of the underlying number system implementation. In the limit, of course, that requires an implementation of exact real arithmetic. Numerical analysis techniques are ultimately needed to ensure the stability of any system based on floating-point numbers.

CML (Concurrent ML) formalized synchronous operations as first-class, purely functional, values called “events” [15]. Our event combinators “`.|.`” and “`==>`” correspond to *CML*’s `choose` and `wrap` functions. There are substantial differences, however, between the meaning given to “events” in these two approaches. In *CML*, events are ultimately used to perform an *action*, such as reading input from or writing output to a file or another process. In contrast, our events are used purely for the values they generate. These values often turn out to be behaviors, although they can also be new events, tuples, functions, etc.

Concurrent Haskell [12] contains a small set of primitives for explicit concurrency, designed around Haskell’s monadic support for I/O. While this system is purely functional in the technical sense, its semantics has a strongly imperative feel. That is, expressions are evaluated without side-effects to yield concurrent, imperative computations, which are executed to perform the implied side-effects. In contrast, modeling entire behaviors as implicitly concurrent functions of continuous time yields what we consider a more declarative feel.

Several languages have been proposed around the *synchronous data-flow* notion of computation. The general-purpose functional language *Lucid* [16] is an example of this style of language, but more importantly are the languages *Signal* [8],

Lustre [4], and *Esterel* [2, 3] which were specifically designed for control of real-time systems. In *Signal*, the most fundamental idea is that of a *signal*, a time-ordered sequence of values. Unlike FRP, however, time is not a value, but rather is implicit in the ordering of values in a signal. By its very nature time is thus discrete rather than continuous, with emphasis on the relative ordering of values in a data-flow-like framework. The designers of *Signal* have also developed a clock calculus with which one can reason about *Signal* programs. *Lustre* is a language similar to *Signal*, rooted again in the notion of a sequence, and owing much of its nature to *Lucid*.

Esterel is perhaps the most ambitious language in this class, for which compilers are available that translate *Esterel* programs into finite state machines or digital circuits for embedded applications. More importantly in relation to our current work, a large effort has been made to develop a formal semantics for *Esterel*, including a constructive behavioral semantics, a constructive operational semantics, and an electrical semantics (in the form of digital circuits). These semantics are shown to correspond in a certain way, constrained only by a notion of stability.

10. ACKNOWLEDGEMENTS

We would like to thank the PLDI referees for their insightful comments and instructive feedback. Also thanks to our funding agencies, NSF through grants CCR-9706747 and CCR-9900957, and DARPA through grant F33615-99-C-3013 administered by the AFOSR.

11. REFERENCES

- [1] Tom M. Apostol. *Mathematical Analysis — A Modern Approach to Advanced Calculus*. Addison-Wesley, 1957.
- [2] Gerard Berry. *The Foundations of Esterel*. MIT Press, 1998.
- [3] Gerard Berry. The constructive semantics of pure esteral (draft version 3). Draft Version 3, Ecole des Mines de Paris and INRIA, July 1999.
- [4] P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice. *Lustre: A declarative language for programming synchronous systems*. In *14th ACM Symp. on Principles of Programming Languages*, January 1987.
- [5] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX, October 1997.
- [6] Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.
- [7] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
- [8] Thierry Gautier, Paul Le Guernic, and Loic Besnard. *Signal: A declarative language for synchronous programming of real-time systems*. In Gilles Kahn, editor, *Functional Programming Languages and*

Computer Architecture, volume 274 of *Lect Notes in Computer Science*, edited by G. Goos and J. Hartmanis, pages 257–277. Springer-Verlag, 1987.

- [9] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.
- [10] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.
- [11] Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.
- [12] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.
- [13] John Peterson, Gregory Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.
- [14] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN, Jan 1999.
- [15] John H. Reppy. CML: A higher-order concurrent language. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.
- [16] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press U.K., 1985.

APPENDIX

A. PROOF OF THEOREMS

We first point out the following useful property that *every* behavior and event observes:

Definition 4. (n-equality) Given an integer n and two lists l_1 and l_2 , if

$$\begin{aligned} \text{length } l_1 &\geq n, \text{ length } l_2 \geq n, \text{ and} \\ \text{take } n \ l_1 &= \text{take } n \ l_2, \end{aligned}$$

then we say that l_1 and l_2 are n -equal, which we write as $l_1 \stackrel{n}{=} l_2$.

LEMMA 1. For any $b \in \langle \text{Behavior}_\alpha \rangle$, $e \in \langle \text{Event}_\alpha \rangle$, $ts, ts' \in \langle \text{Time} \rangle$, and $n \in \mathbb{N}$, $ts \stackrel{n}{=} ts'$ implies that:

$$\begin{aligned} [b] \ ts &\stackrel{n}{=} [b] \ ts', \text{ and} \\ [e] \ ts &\stackrel{n}{=} [e] \ ts'. \end{aligned}$$

This Lemma essentially says that the current value of a behavior or event does not depend on the future. The proof of this lemma is an induction on the syntactic structure of an FRP expression.

LEMMA 2. $[\text{lift1}] \ f \ b \ ts = \text{map } f \ (b \ ts)$.

PROOF.

$$\begin{aligned} &[\text{lift1}] \ f \ b \ ts \\ &= ([\text{lift0}] \ f \ [\$*] \ b) \ ts \quad (\text{definition of lift1}) \\ &= (\lambda ts. \text{zipWith } [(\$)] \ ([\text{lift0}] \ f \ ts) \ (b \ ts)) \ ts \\ &\quad (\text{definition of } \$*) \\ &= \text{zipWith } [(\$)] \ ([\text{lift0}] \ f \ ts) \ (b \ ts) \\ &= \text{zipWith } [(\$)] \ (\text{map } (\text{const } f) \ ts) \ (b \ ts) \\ &\quad (\text{definition of lift0}) \\ &= \text{map } f \ (b \ ts) \end{aligned}$$

□

LEMMA 3. $[\text{lift2}] \ f \ b_1 \ b_2 \ ts = \text{zipWith } f \ (b_1 \ ts) \ (b_2 \ ts)$.

The proof is not hard and omitted.

Theorem 1

PROOF. For any $\epsilon > 0$, let $\delta = 1$, for any P_T^t such that $|P_T^t| < \delta$, we have

$$\begin{aligned} &\tilde{at}[\text{time}] \ P_T^t \\ &= \text{last } ([\text{time}] \ P_T^t) \quad (\text{definition of } \tilde{at}) \\ &= \text{last } P_T^t \quad (\text{definition of time}) \\ &= t \end{aligned}$$

Hence $|\tilde{at}[\text{time}] \ P_T^t - t| = 0 < \epsilon$. □

Theorem 2

PROOF. For any $\epsilon > 0$, let $\delta = 1$, for any P_T^t such that $|P_T^t| < \delta$, we have

$$\begin{aligned} &\tilde{at}[\text{lift0 } c] \ P_T^t \\ &= \text{last } ([\text{lift0 } c] \ P_T^t) \\ &= \text{last } ([\text{lift0}] \ [c] \ P_T^t) \\ &= \text{last } (\text{map } (\text{const } [c]) \ P_T^t) \quad (\text{definition of lift0}) \\ &= [c] \end{aligned}$$

Hence $|\tilde{at}[\text{lift0 } c] \ P_T^t - [c]| = 0 < \epsilon$. □

Theorem 3

PROOF.

$$b_t = \lim_{|P_T^t| \rightarrow 0} \tilde{at}[b] \ P_T^t$$

Thus

$$\begin{aligned} &[f] \ b_t \\ &= [f] \ \left(\lim_{|P_T^t| \rightarrow 0} \tilde{at}[b] \ P_T^t \right) \\ &= \lim_{|P_T^t| \rightarrow 0} [f] \ (\tilde{at}[b] \ P_T^t) \\ &\quad ([f] \text{ is continuous at } \lim_{|P_T^t| \rightarrow 0} \tilde{at}[b] \ P_T^t) \\ &= \lim_{|P_T^t| \rightarrow 0} [f] \ (\text{last } ([b] \ P_T^t)) \end{aligned}$$

$$\begin{aligned}
&= \lim_{|P_T^t| \rightarrow 0} \text{last} (\text{map } \lfloor f \rfloor (\lfloor b \rfloor P_T^t)) \\
&= \lim_{|P_T^t| \rightarrow 0} \text{last} (\lfloor \text{lift1} \rfloor \lfloor f \rfloor \lfloor b \rfloor P_T^t) \\
&\quad (\text{lemma 2}) \\
&= \tilde{at}^* \lfloor \text{lift1 } f \rfloor T t
\end{aligned}$$

□

Theorem 4

PROOF. For any $\epsilon > 0$, since $\lfloor f \rfloor$ is uniformly continuous, there is an $\eta > 0$, such that for any v, v' , $|v - v'| < \eta$ implies $|\lfloor f \rfloor v - \lfloor f \rfloor v'| < \epsilon$.

Since $\tilde{at} \lfloor b \rfloor P_T^t \mapsto fb(t)$ uniformly on S , for the above η , there is a $\delta > 0$, such that for every $t \in S$, $|P_T^t| < \delta$ implies $|\tilde{at} \lfloor b \rfloor P_T^t - fb(t)| < \eta$.

Hence for every $t \in S$, $|P_T^t| < \delta$ implies

$$\begin{aligned}
&|\tilde{at} \lfloor \text{lift1 } f \rfloor P_T^t - \lfloor f \rfloor (fb(t))| \\
&= |\lfloor f \rfloor (\tilde{at} \lfloor b \rfloor P_T^t) - \lfloor f \rfloor (fb(t))| \\
&< \epsilon
\end{aligned}$$

□

Corollary 1

PROOF. Since

$$\tilde{at} \lfloor \text{time} \rfloor P_T^t \mapsto t \text{ uniformly on } (-\infty, \infty)$$

(by theorem 1), and $\lfloor \sin \rfloor = \sin$ is uniformly continuous, we have

$$\tilde{at} \lfloor \text{lift1 } \sin \text{ time} \rfloor P_T^t \mapsto \sin t$$

uniformly on $(-\infty, \infty)$ (by theorem 4).

According to the definition of lifted \sin :

> instance Floating a => Floating (Behavior a) where
> sin = lift1 sin

we have

$$\tilde{at} \lfloor \sin \text{ time} \rfloor P_T^t \mapsto \sin t \text{ uniformly on } (-\infty, \infty).$$

□

Theorem 7

PROOF. For any $\tau \in [T, t]$,

$$\begin{aligned}
&\lim_{|P_T^\tau| \rightarrow 0} \tilde{at} \lfloor \text{integral } f \rfloor P_T^\tau, \text{ where } P_T^\tau = [t_0, t_1, \dots, t_n]. \\
&= \lim_{|P_T^\tau| \rightarrow 0} \text{last} (\lfloor \text{integral } f \rfloor P_T^\tau) \\
&= \lim_{|P_T^\tau| \rightarrow 0} \sum_{i=1}^n (\lfloor f \rfloor P_T^\tau)_{(i)} \cdot \Delta t_i, \quad \text{where } \Delta t_i = t_i - t_{i-1}. \\
&\quad (\text{definition of integral}) \\
&= \lim_{|P_T^\tau| \rightarrow 0} \sum_{i=1}^n (\text{last} (\lfloor f \rfloor \tilde{P}_i)) \cdot \Delta t_i, \\
&\quad \text{where } \tilde{P}_i = \text{take } i P_T^\tau. \quad (\text{lemma 1})
\end{aligned}$$

$$= \lim_{|P_T^\tau| \rightarrow 0} \sum_{i=1}^n (\tilde{at} \lfloor f \rfloor \tilde{P}_i) \cdot \Delta t_i$$

Furthermore,

$$\begin{aligned}
&\left| \lim_{|P_T^\tau| \rightarrow 0} \sum_{i=1}^n (\tilde{at} \lfloor f \rfloor \tilde{P}_i) \cdot \Delta t_i - \int_T^\tau (\tilde{at}^* \lfloor f \rfloor T \eta) d\eta \right| \\
&= \left| \lim_{|P_T^\tau| \rightarrow 0} \sum_{i=1}^n (\tilde{at} \lfloor f \rfloor \tilde{P}_i) \cdot \Delta t_i - \right. \\
&\quad \left. \lim_{|P_T^\tau| \rightarrow 0} \sum_{i=1}^n (\tilde{at}^* \lfloor f \rfloor T t_i) \cdot \Delta t_i \right| \\
&= \left| \lim_{|P_T^\tau| \rightarrow 0} \sum_{i=1}^n (\tilde{at} \lfloor f \rfloor \tilde{P}_i - \tilde{at}^* \lfloor f \rfloor T t_i) \cdot \Delta t_i \right| \\
&\leq \lim_{|P_T^\tau| \rightarrow 0} \sum_{i=1}^n e_i \cdot \Delta t_i, \\
&\quad \text{where } e_i = |\tilde{at} \lfloor f \rfloor \tilde{P}_i - \tilde{at}^* \lfloor f \rfloor T t_i|.
\end{aligned}$$

Since $\tilde{at} \lfloor f \rfloor P_T^\tau \mapsto \tilde{at}^* \lfloor f \rfloor T \tau$ uniformly converges on $[T, t]$, for every given $\epsilon > 0$, there is a $\delta > 0$, such that as long as $|P_T^\tau| < \delta$,

$$e_i < \epsilon, \text{ for every } 1 \leq i < \text{length } P_T^\tau.$$

Thus when $|P_T^\tau| < \delta$, for every $\tau \in [T, t]$ we have

$$\sum_{i=1}^n e_i \cdot \Delta t_i \leq \sum_{i=1}^n \epsilon \cdot \Delta t_i = \epsilon \cdot (\tau - T) \leq \epsilon \cdot (t - T)$$

Since ϵ is arbitrary, $\tilde{at} \lfloor \text{integral } f \rfloor P_T^\tau$ uniformly converges to $\int_T^\tau (\tilde{at}^* \lfloor f \rfloor T \eta) d\eta$. □

Theorem 10

PROOF. If $\widetilde{occ}^*[e] T t = []$, then there is a $\delta > 0$ such that for every partition P_T^t of $[T, t]$ where $|P_T^t| < \delta$, $\text{justValues } P_T^t (\lfloor e \rfloor P_T^t) = []$. Therefore

$$\begin{aligned}
&\lim_{|P_T^t| \rightarrow 0} \text{last} (\lfloor b \text{ 'until' } e \rfloor P_T^t) \\
&= \lim_{|P_T^t| \rightarrow 0} \text{last} (\lfloor b \rfloor P_T^t) \\
&\quad (\text{definition of until}) \\
&= \tilde{at}^* \lfloor b \rfloor T t
\end{aligned}$$

If $\widetilde{occ}^*[e] T t = [(t_1, \lfloor b_1 \rfloor), \dots, (t_m, \lfloor b_m \rfloor)]$, where $m > 0$, then for a partition $P = [\eta_0, \eta_1, \dots, \eta_n]$ of $[T, t]$, when $|P|$ is small enough, $\text{justValues } P (\lfloor e \rfloor P)$ will have m elements. Let the first of the m elements be $(\eta_k, \lfloor b' \rfloor)$, and $\tilde{P}^k = [\eta_k, \eta_{k+1}, \dots, \eta_n]$, then

$$\begin{aligned}
&\lim_{|P| \rightarrow 0} \text{last} (\lfloor b \text{ 'until' } e \rfloor P) \\
&= \lim_{|P| \rightarrow 0} \text{last} (\lfloor b' \rfloor \tilde{P}^k) \\
&\quad (\text{definition of until}) \\
&= \lim_{\eta \rightarrow t_1} \tilde{at}^* \lfloor b_1 \rfloor \eta t
\end{aligned}$$

□

DALI: An Untyped, CBV Functional Language Supporting First-Order Datatypes with Binders

(Summary*)

Emir Pašalić, Tim Sheard, Walid Taha†

ABSTRACT

Writing (meta-)programs that manipulate other (object-) programs poses significant technical problems when the object-language itself has a notion of binders and variable occurrences. Higher-order abstract syntax is a representation of object programs that has recently been the focus of several studies. This paper points out a number of limitations of using higher order syntax in a functional context, and argues that DALI, a language based on a simple and elegant proposal made by Dale Miller ten years ago can provide superior support for manipulating such object-languages. Miller's original proposal, however, did not provide any formal treatment. To fill this gap, we present both a big-step and a reduction semantics for DALI, and summarize the results of our extensive study of the semantics, including the rather involved proof of the soundness of the reduction semantics with respect to the big-step semantics. Because our formal development is carried out for the untyped version of the language, we hope it will serve as a solid basis for investigating type system(s) for DALI.

1. INTRODUCTION

Programs are data. Nothing makes this point stronger than the ever increasing need for reliable programs with verified properties. As software systems become more complex, and play increasingly important roles in critical systems there is an ever increasing need for optimizing, analyzing, verifying and certifying software.

Each one of these tasks involves automatic manipulation

*The complete technical development appears in a technical report available online. This paper focuses on the describing DALI from the point of view of language design and programming.

†Sheard and Pašalić are supported by the USAF Air Materiel Command, contract # F19628-96-C-0161, NSF Grants CCR-9803880 and IRI-9625462, and the Department of Defense. Taha is supported by a Postdoctoral Fellowship, funded by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403.

of programs or, *meta-programming*. As with any kind of programming, effective meta-programming relies heavily on the presence of the appropriate support from the (meta-) programming language. The goal of this paper is to advocate a novel approach to representing programs in a manner superior to the main contenders available today. Our approach gives rise to a simple equational theory that can be used to reason about the program equivalence of meta-programs.

1.1 Meta-Programming as Programming

It is our thesis that traditional programming language techniques, including those from the operational, categorical, axiomatic, and denotational traditions can be applied equally effectively to meta-programming languages [45]. In many instances, this means that the technical challenge is “internalizing” various meta-level operations, such as quotation [49], evaluation [48; 29; 4; 45; 47], and type analysis [44; 46], into a formal programming language, and subjecting them to the same high standards developed by the semantics community. This approach has numerous pragmatic benefits, including:

1. We succeed in magnifying the subtle features of the operations under investigation, and, often times, in addressing them in a systematic and complete manner. From the software engineering point of view, this translates into enhanced safety and reliability.
2. We succeed in assigning a uniform semantics to these operations that must otherwise be carried out in an *ad hoc* fashion. This can be done to the extent that we can provide mathematically verified reasoning principles for these operations in the form of equational theories. From the software engineering point of view, this translates into enhanced correctness.
3. We make these operations available to the programmer in a uniform way, thus providing more him or her with more control over the *behavior* of the system. From the software engineering point of view, this translates into enhanced predictability.

1.2 Synthesis vs. Analysis

There are two different kinds of program manipulation: *program synthesis*, and *program analysis*. The combination of the two is necessary for expressing general *program transformations*. In what follows we outline the state of the art in language support for both synthesis and analysis, and ex-

plain how the present work on DALI fits in the context of analysis.

1.2.1 Synthesis and Multi-Stage Programming

Many recent studies have concentrated on language level support for program synthesis: works on multi-level [12; 11; 30; 28] and multi-stage [49; 48; 29; 4; 45; 47] programming languages have investigated basic problems relating to language support needed for program synthesis such as how to build program fragments, how to combine smaller program fragments into larger ones, and how to execute such fragments in a user friendly, hygienic, and type-safe manner. But while multi-stage programming constructs provide good support for the construction and execution of object-code, they provide no support for analysis. In fact, adding constructs for analyzing code fragments can severely weaken the notion of observational equivalence in such languages [47].

1.2.2 Analysis and Higher Order Syntax

In contrast, substantially fewer studies have focused on language level support for program analysis [21; 39; 13]. With few exceptions (see for example Bjorner [5]), the most popular tool for these studies has been *higher order abstract syntax* [38] (HOAS), and have taken place in the context of logic programming languages [1]. In the remainder of this paper we shall (without drawing too fine a distinction) refer to all approaches to syntax that represent object-level binding constructs by meta-language binding constructs in a uniform way as *higher-order abstract syntax*.

A program analysis inspects the *structure* and environment of an object-program and computes some value as a result. Results can be data- or control-flow graphs, or even another object-program with properties based on the properties of the source object-program. Examples of these kinds of meta-systems are: program transformers, optimizers, and partial evaluation systems [22].

Program analyses are particularly difficult to write correctly if they must manipulate terms that have a notion of statically scoped variables. The exact representation of the variable is generally uninteresting, and often requires subtle administrative changes so that it maintains its original "meaning".

The primary example of such administrative changes is a renaming when the "direct" representation of variables is used, and "shift" and "lift" operations when de Bruijn indices are used. The first representation relies, typically, on the use of state, a "gensym" operation, and the second representation is generally considered "too human unfriendly". Because of this, representing object-programs using first order algebraic data structures which use strings or other atomic values to represent variables are notoriously hard to manipulate correctly.

A more pressing concern is that implementing such operations once is not enough: They need to be implemented for *each object-language that has binding constructs*. The basic problem is therefore pervasive, it appears in almost every interesting language.

The basic idea that we advocate is to (uniformly) exploit the binding mechanism of the meta-language to implement the binding mechanism(s) of the object-language, i.e. use

functions in the meta-language to implement binding in the object-language. At first glance, this looks like a promising idea, but a number of subtle problems arise. We explicate these problems carefully in Section 3. The problems arise because the functions of the meta-language have two properties which, while necessary for their use as functions, get in the way of their use as binding mechanisms. These properties are: extensionality and delayed computation. Extensionality means that one cannot observe the structure of a function, other than by applying it to get a result. Delayed computation means that computations embodied in a function do not occur until the function is applied. What we need is a new kind of binding, without these properties.

In this paper, we develop such a binding mechanism by refining some ideas of Dale Miller's [23]. This new binding mechanism can be incorporated into a functional language with first-order datatypes, and together they can be used to represent variable binding in object-languages. This mechanism can be systematically reused. In addition, we develop a sound syntactic system for reasoning about the equivalence of functional programs that use this new binding mechanism.

1.3 Contribution

The contribution of this paper is simple and focused: a call-by-value operational semantics for an untyped functional programming language with an extension that supports first-order datatypes (FOD) with binders.

We have applied the rigorous standards of language design and semantic analysis to both the host language (the lambda calculus) and the extension and discovered that the two are mutually compatible. The combined language enjoys a non-trivial equational theory where beta convertibility is a congruence, and is therefore unlikely to invalidate known optimizations for a call-by-value functional language.

We believe that our present operationally-based study complements the recent model-theoretic approach of Gabbay and Pitts [17], Hofmann [20], and Fiore, Plotkin, and Turi [16]. For example, whereas Pitts and Gabbay's recent work emphasizes that a type system is required for their language to ensure that "namefulness" doesn't spread everywhere, our language is untyped, and does not appear to give rise to any non-standard "namefulness" problems.

2. HOAS V.S. FIRST ORDER DATATYPES

The precise semantics of (*meta*-)programs depends crucially on the basic properties of the *representation* of object-programs. This question of representation is the focus of the present study.

The essence of the representation we propose goes back at least to Church [8]. The idea is to exploit the binding mechanism of the meta-language to implement the binding mechanism(s) of the object-language. This is also the essence of Pfenning and Elliot [38] and Miller' [23; 25; 26] *higher-order syntax* (HOAS) representation. To illustrate the basic idea of higher-order syntax, consider the definitions of Term and Term' below.


```

data Term
= App Term Term
| Abs String Term
| Const Int
| Var String

data Term'
= App' Term' Term'
| Abs' Term' -> Term'
| Const' Int

```

In `Term'` we represent the object-language lambda abstraction (`Abs'`) using the meta-language function abstraction. This way, functions such as `id` and `app` are represented by applying the `Abs'` constructor to a meta-language function:

```

-- \ x -> x
id = Abs "x" (Var "x")

-- \f -> \ x -> f x
app = Abs "f"
      (Abs "x"
        (App (Var "f")
              (Var "x")))

-- \ x -> x
id' = Abs' (\ x -> x)

-- \f -> \ x -> f x
app' = Abs' (\ f ->
              Abs' (\ x ->
                    (App' f x)))

```

The HOAS representation (`Term'`) is elegant in that a concrete representation for variables is not needed, and that it is not necessary to invent unique, new names when constructing lambda-expressions which one can only “hope” don’t clash with other names.

3. CRITIQUE OF HOAS

This flavor of HOAS seems like a great idea at first, but careful inspection reveals a few anomalies. It works fine for constructing statically known representations, but quickly breaks down when trying to construct or observe a representation in an algorithmic way. We quickly provide a few small examples that illustrate the problems we have encountered.

P_1 Opaqueness: We cannot pattern match or observe the structure of the body of an `Abs'`, or any object-level binding, because they are represented as functions in the meta-language, and meta-level functions are extensional.

We can observe this by casting our `Term'` example above into a real program formulated in ML, and noticing that `id` prints as `Abs' fn`.

(* Actual ML Program Execution *)

```

- datatype Term'
  = App' of (Term' * Term')
  | Abs' of (Term' -> Term')
  | Const' of int;

- val id = Abs' (fn x => x);
val id = Abs' fn : Term'

```

P_2 Junk [7; 6]: I.e., there are terms in the meta-language with type `Term'` that do not represent any legal object-program. Consider:

```

junk = Abs' (\ x -> case x of App' f y -> y
                          ; Const' n -> x
                          ; Abs' _ -> x)

```

No legal object-program behaves in this way.

P_3 Latent Divergence: Because functions delay computation, a non-terminating computation producing a `Term'` may delay non-termination until the `Term'` object is observed. This may be arbitrarily far from its construction, and can make things very hard to debug. Consider the function `bad` below:

```

bad (Const' n) = Const' (n+1)
bad (App' x y) = App' (bad x) (bad y)
bad (Abs' f) = Abs' (\ x -> diverge (bad (f x)))

```

`bad` walks over a `Term'` increasing every explicit constant by one. Suppose the programmer made a mistake and placed an erroneous divergent computation in the `Abs'` clause. Note that `bad` does not immediately diverge.

P_4 Expressivity: Using HOAS, there exist (too many) meta-functions over object-terms that cannot be expressed. Consider writing a `show` function for `Term'` that turns a `Term'` into a string suitable for printing.

```

show (App' f x) = (show f) ++ " " ++ (show x)
show (Const' n) = toString n
show (Abs' f) = "\ " ++ "?v ++ " -> "
                ++ (show (f ?v))

```

What legal meta-program value do we use for `?v`? We need some sort of “variable” with type `Term'` but no such thing can be created. There are “tricks” for solving this problem [14], but in the end, they only make matters worse.

Our approach to these problems is to cast our search for solutions as an exercise in programming language design. The following subsections offer an informal discussion of each problem and a potential solution by the introduction of additional language features, and provide examples of how these language features might be used. Our biggest challenge is to discover features that interact well, both with each other, and with the existing features of the language we wish to add them to.

3.1 Opaqueness

To solve the opaqueness problem a number of researchers have investigated the use of higher-order pattern matching [32]. The basic idea is that programmers use a higher-order interface to the object-language because it is expressive and easy to use, but the actual underlying implementation is first order.

One tries to supply an enriched interface that gives programmers access to this first-order implementation in a safe manner, that still supports all the benefits of a higher-order implementation. To illustrate this consider the (not necessarily semantics-preserving) rewrite rule `f` for object-terms `Term'`, which might be expressed as:

$f : (\lambda x. (e' 0)) \rightarrow (e' [0/x])$.

Here, we use the notation, that a primed variable is a meta-variable. Thus e' is a meta-variable of the rule, and $e' [0/x]$ indicates the capture free substitution of 0 for x in e' .

Higher-order pattern matching is a programming language mechanism, that allows us to express that we wish to observe the inner structure of meta-language abstraction, and that parts of the body of this abstraction (i.e. e') may have free occurrences of x inside.

We use a higher-order pattern when we wish to analyze the structure of a constructor like `Abs'` which takes a meta-function as an argument. Like all patterns, a higher order

pattern “binds” a meta-variable. The meta-variable bound by a higher-order pattern does not bind to an object-term, but instead binds to a function. This function captures the subtlety that e' might have free occurrences of x . Given the bound variable, as input, it reconstructs the body of the abstraction. Given a term as input, it substitutes the term for each free occurrence of the bound variable in the body.

The bound meta-variable is a function from $\text{Term}' \rightarrow \text{Term}'$. We make this language mechanism concrete by extending the notion of pattern in our meta-language. Patterns can now have explicit lambda abstractions, but any pattern-variables inside the body of the lambda abstraction are higher-order pattern-variables, i.e. will bind to functions. Consider below, an example implementing the rewrite rule above.

```
f (Abs'(\ x -> App'(e' x)(Const 0))) = e'(Const 0)
f x = x
```

In this example the meta-function f matches its argument against an object-level abstraction ($\text{Abs}' \dots$) using an object-level pattern. The pattern specifies that the body of the matched abstraction must be an application (App') of a function term ($e' x$) to a constant ($\text{Const } 0$). The function part of this object-application can be any term. This term may have free occurrences of the object-bound variable (which we write as x in the pattern, but which can have any name in the object-term it matches against). Because of this we use a higher-order pattern ($e' x$) which applies e' to x to indicate that e' is a function whose argument is the object-bound variable x .

This extension differs from normal pattern matching, in that neither meta-level abstractions ($\lambda x \rightarrow \dots$) nor applications of meta-variables ($e' x$) normally appear in regular patterns.

If the underlying implementation is first order (like Term), patterns of this form have an efficient and decidable implementation. The clause

```
f (Abs'(\ x -> App'(e' x)(Const 0))) = e' (Const 0)
```

would translate into an implementation using Term as follows:

```
f (Abs x (App e (Const 0))) =
  let e' y = subst [(x,y)] e
  in e' (Const 0)
```

The key advantage of this approach is that users get to use the expressive and safe HOAS interface, and the substitution function need not be written by the programmer but can be supplied by the underlying implementation.

The solution of using (a hidden) underlying first order implementation, but supplying a higher-order interface, extends nicely to term construction as well as term observation.

A construction like: $(\text{Abs}' f) :: \text{Term}'$ could be translated into an underlying implementation based on first-order, observable, data-structures (i.e. Term) by using a *gensym* construct to provide a “fresh” name for the required object-bound variable:

```
let y = gensym () in Abs y (f (Var y)).
```

Again both the *gensym* and the underlying first-order implementation is hidden from the user.

3.2 Junk

Junk is a serious problem in that it allows meta-programs to represent non-existent terms in the object-language. Junk arises because the body of an object-binding is a computation (i.e. a suspended function), rather than a constant piece of data. This causes two kinds of problems:

1. The computation can “observe” the bound variable, and do ill-advised things like pattern matching. A valid object-binding only “builds” new structure around the variable. It does not observe the bound variable.
2. The computation can introduce effects. In this case the computational effects of the meta-language, such as nontermination, are introduced into the purely syntactic representation of the object language. Even worse, the effects are only introduced when the object-term is observed. If a term is observed multiple times, it causes the effects to be introduced multiple times.

3.3 Latent Divergence

So we see that that junk and latent divergence are really two facets of the same problem. To fix these problems we need a binding construct which preserves static scoping (like normal meta-level functions) but which does *not* delay computation. What we need is a binding construct which forces computation “under the lambda” [45].

Ten years ago, Dale Miller proposed a new meta-level binding construct for implementing HOAS in ML [23] which did exactly this. He introduced a new binary type constructor ($a \Rightarrow b$) which names the type of an object level binding of a terms in b terms. The new type constructor was used in place of the function type constructor to denote object-level abstraction.

We introduce DALI, a language based upon a refinement of Miller’s idea. We compare it to HOAS, and illustrate its intended use by a number of examples. In DALI, the object-binding mechanism is separate from the function construct of the meta-language. This allows us to restrict the range of junk, and the introduction of erroneous effects. Consider our small lambda calculus example once again.

```
datatype Term
  = App Term Term
  | Abs Term => Term
  | Const Int
```

Terms of type $a \Rightarrow b$ are introduced using the meta-language construct for object-binding introduction. The expression level syntax of the meta-language, is analogous to the syntax of the type constructor for object-level bindings. For example: $(\#x \Rightarrow \text{App}(\#x, \text{Const } 0)) :: (\text{Term} \Rightarrow \text{Term})$. Here we use the hash ($\#x$) notation to distinguish object-level variables from meta-level variables. An important property of object-variables, is that they cannot escape their scope. Like meta-level function binding, object-binding respects static scoping. The \Rightarrow introduction construct $(\#x \Rightarrow e)$ delimits the scope of $\#x$ to e . The key

property of object-binding is that evaluation proceeds under \Rightarrow .

Below are two different examples of constructing an object-language program. The first using meta-level functions as the binding mechanism, and the second using object-level abstraction:

```
Abs'(\x -> bottom)           Abs(#x => bottom)
```

The expression on the left uses a meta-language binding mechanism (λ abstraction). It succeeds in constructing a representation of an object-language program which obviously has no meaning. The expression on the right, however, does not represent any object-language program, since the expression never terminates. Note that the effect on the left has seeped into the object-language program representation (junk), while on the right non-termination occurs before the object-language program is constructed and thus is never present in the object-language program itself.

A more sophisticated example is the copy function over Term

```
copy (App f x) = App (copy f) (copy x)
copy (Const n) = Const n
copy (Abs(#x => e' #x)) = Abs(#y => (copy (e' #y)))
copy (x @ #_) = x
```

To those familiar with functional programming, the first two clauses should be clear. The third clause, uses the higher-order pattern matching introduced earlier, only applied here in the context of the new object-level binding construct. Since evaluation passes under \Rightarrow diverging computations will not delayed.

The fourth clause of the copy function is an artifact of the object-level binding mechanism. Object-variables ($\#x$) introduced using the object-binding syntax: $(\#x \Rightarrow \dots)$, are a new type of constant. The actual name of such a constant is not accessible to the programmer. There are two operations that are necessary on object-variables, it should be possible to distinguish them from other object-terms, and it should be possible to compare them using equality, in order to tell them apart.

Thus, functions over object-languages, must have a clause for object-bound variables. Object-bound variables are distinct from all other constructors, and are common to all object-languages. The pattern $\#_.$ matches any object-variable, but fails to match other constructors. The binding says nothing about the name of the variable it binds to. The notation $(x @ \#_.)$ introduces a meta-level variable x , bound to the object-level variable matched by the object-pattern $\#_.$

3.4 Expressivity

It is sometimes necessary to eliminate object-bound variables. This is done in one of two ways. First by applying a higher-order pattern variable to some value $x :: \text{Term}$, the occurrences of the bound variable will be replaced with x .

This is not always sufficient since it does not provide any way of transforming a object-binding into anything other than another object-binding. This was the problem with the show function (Section 3). This is why HOAS using

meta-level function binding cannot express some functions. Object-level binding allows us to solve this problem.

The solution is a new language construct *discharge*. The construct $(\text{discharge } \#x \Rightarrow e1)$ introduces a new object-level variable ($\#x$), whose scope is the body $e1$. The value of the discharge construct is its body $e1$. The body $e1$ can have any ground type, unlike an object-level binding $(\#x \Rightarrow e2)$, where $e2$ must be an object term.

In addition, discharge incurs an obligation that the object variable ($\#x$) does not appear in the value of the body ($e1$). An implementation must raise an error if this occurs.

For example consider a function which counts the number of Const subterms in a Term.

```
count :: Term -> Int
count (Const _) = 1
count (App f x) = (count f) + (count x)
count (Abs(#x => e' #x)) =
    discharge #y => count (e' #y)
count #_ = 0
```

Note how that the fourth clause conveniently replaces all introduced object-bound variables with 0, thus guaranteeing that no object-variable appears in the result. The obligation that the variable does not escape the body of the discharge construct may require a run-time check (though in this example, since the result has type Int, no such occurrence can happen).

If a programmer needs to treat individual object-bound variables in different ways, he can use an environment parameter. Consider the program below, which is the correct implementation of the function show.

```
show x = sh n [] x
where
  sh n (App f x) = (sh n f) ++ " " ++ (sh n x)
  sh n (Const n) = toString n
  sh n (x @ #_) = lookup x n
  sh n (Abs(#y => f #y)) =
    let x = len n
        v = x ++ (toString x)
    in discharge #x =>
        "\\ " ++ v ++ " -> "
        ++ (show ((#x,v):n) (f #x))
```

Here the environment n is a list of pairs mapping object-variables to strings. If the sh function is applied to an object-variable it looks up its name in the environment. For an object-abstraction, $(\text{Abs}'(\#x \Rightarrow f \#x))$, discharge introduces a new object-variable, adds it to the environment, and then applies the higher-order pattern variable f to the introduced variable, and recursively produces a string as the representation of the abstraction's body.

Another example transforms a Term into its de Bruijn equivalent form.

```
data DB
  = DApp DB DB
  | DAbs DB
  | DVar Int
  | DConst Int

DeBruijn env (App f x) =
  DApp (DeBruijn env f) (DeBruijn env x)
```

```

DeBruijn env (Abs(#x => e' #x)) =
  discharge #y => DAbs(DeBruijn (ext env #y)(e' #y))
  where ext env v u =
    if v=u then 0 else 1 + (env u)
DeBruijn env (Const n) = DConst n
DeBruijn env (z @ #_) = env z

```

4. EXAMPLES

In this section we use our language to express some classic manipulations on object-languages.

- Lambda calculus syntax

```

datatype Lterm = App Lterm Lterm
               | Abs Lterm => Lterm
               | Const Int
               | Prod Lterm Lterm

```

- Call-by-name Big-step evaluator for untyped lambda calculus:

```

eval : Lterm -> Lterm
eval (Abs body) = Abs body
eval (App t1 t2) =
  case eval t1 of
    (Abs (#x => body #x)) -> eval (body t2)
eval (Const n) = Const n
eval (Prod x y) = Prod (eval x) (eval y)
eval (x @ #_) = x

```

- CBN lambda calculus (single step) reduction:

```

beta : Lterm -> Lterm -> Lterm
beta (Abs (#z => body #z)) t2 = body t2

```

- Complete development:

```

compdev : Lterm -> Lterm
compdev (Abs(x# => body #x))
  = Abs(#w => compdev (body #w))
compdev (App (Abs(#x => body #x)) y)
  = sub (Abs(#w => compdev (body #w))) (compdev y)
  where sub (Abs(#z => e #z)) x = e x
compdev (App f x) = App(compdev f)(compdev x)
compdev (Prod x y) = Prod(compdev x)(compdev y)
compdev (Const n) = Const n
compdev (x @ #_) = x

```

- Substitution on Lterms.

```

find x [] = Nothing
find x ((y,v):ys) = if x==y
  then v
  else find x ys

subst: Lterm -> [(Lterm,Lterm)] -> Lterm
subst x env =
  case find x env of
    Just t -> t
    Nothing ->
      case x of
        v @ #_ -> v
        Abs(#x => e #x) ->
          Abs(#w => subst env (e #w))
        App x y -> App(subst env x)(subst env y)
        Prod x y -> Prod(subst env x)(subst env y)
        Const n -> Const n

```

5. NEW FEATURES OF DALI

The language DALI contains some features that behave in untraditional ways. It is useful to call attention to these features.

- *Object variable bindings*: Unlike the meta-language binding construct, the evaluation of an object-level abstraction $(\lambda x \Rightarrow e)$ proceeds “under” the \Rightarrow .

- *Ground (or equality) values*: such values can be compared for simple structural equality. The important property of ground values is that they do not contain functions. Only ground values are used to represent valid object languages.

In order to compare object-language terms for equality it is necessary to compare object-variables for equality. This must be a primitive in the language. Equality on object-language types is important for two reasons. First, it facilitates an important programming technique, illustrated in our de Bruijn notation example above. Second, it makes possible higher-order pattern matching (see below).

- *Object-variable matching*: Comparing object-language terms for equality is not enough for the meta-programs in our examples. We must be able to distinguish object-level variables from other object-level terms. This is the purpose of the $(\#_)$ pattern.

- *Higher-order pattern matching*: A higher order pattern variable (i.e. x in $(\lambda(\lambda z \Rightarrow x \#z) \rightarrow e)$ is bound to a (meta-level) function that returns the body of the object-level abstraction after replacing the object-variable with its argument. This in effects internalizes substitution for bound object-level variables in object programs. In order to implement such a scheme, it is important that the object-abstraction body be an equality type. I.e. we must somehow disallow types of the form $(a \Rightarrow (b \rightarrow c))$. If we do not do this then interesting anomalies may occur. For example consider:

```

f (#z => x #z) = x (Const 0)
w = (#x => (\ y -> Prod #x (Const 5)))

```

If we apply f to w , we must build a meta-level function x which replaces all occurrences of $\#x$ in $(\lambda y \rightarrow \text{Prod } \#x \text{ (Const 5)})$ with (Const 0) . It is unlikely we can do this if functions are only extensional.

6. A NOTE ABOUT “DISCHARGE”

In defining meta-programs, the use of discharge is often crucial, since it allows for eliminating an object-level binding and performing computations only on its body. However, the binding's body can be safely extracted only if there is a guarantee that a heretofore bound object-level variable cannot become free as a result of computation over its body. There are two ways of adding discharge to DALI: First, as a new language construct with appropriate reduction rules; and second, as a function defined by the user on a per-datatype basis.

In the present paper, we opt for the second design decision in order to keep the core calculus of DALI, and its technical development as small as possible. We present an example of a user defined discharge function for the lambda-term datatype (Lterm):

```

discharge (#w => t #w) =
  case (#a => find #a (t #a)) of
    (#z => True ) -> t ()
    (#z => False) -> diverge

find var (App t1 t2) =
  (find var t1) or (find var t2)
find var (Abs(#w=>b #w)) =
  case (#z => find var (b #z)) of
    (#z => True ) -> True
    (#z => False) -> False
find var (Prod t1 t2) =
  (find var t1) or (find var t2)
find var (Const n) = False
find var (x @ #_) =
  if x = var then True
  else False

```

The function `discharge` simply searches the body of an object-level abstraction for the abstracted object-level variable. If the variable is not found in the body, the program simply returns the body itself. Otherwise, the computation diverges.

It is important to note that in DALI, `discharge`, whether added as a language construct or defined as a function, can be used only on ground values, i.e., values that do not contain suspended computation. Extending `discharge` to apply to values that contain abstraction causes confluence and soundness problems similar to those described in section 8.1.

However, there appear to be situations where such a more general version `discharge` is desirable. The example below, implements a kind of evaluation for the familiar encoding of untyped lambda terms, using an environment.

```

data Value = Vint Int
            | Vprod Value Value
            | Vfun Value -> Value

type Env = [(Lterm * Value)]

eval' : Env -> Lterm -> Value
eval' env (Abs (#x => b #x)) =
  discharge #w =>
    Vfun(\ y -> eval' (extend #x y env) (b #w))
eval' env (App t1 t2) =
  case eval' env t1 of
    Vfun f -> f (eval' env t2)
eval' env (Const n) = Vint n
eval' env (Prod x y)
  = Prod(eval' env x)(eval' env y)
eval' env (x @ #_) = env x

```

In the present version of the language it is impossible to define a `discharge` function needed for the second clause of `eval'`, since it would involve detection of free object-bound variables in a term that contains an (extensional) meta-level function. On the other hand, the example is intuitively correct, and one can convincingly argue from the

definition of the function `eval'` that the discharged object-level variables indeed never do appear in the values of `eval'`. Whether an appropriate mechanism can be introduced to extend `discharge` to such cases remains a question yet to be fully addressed for DALI.

7. FORMAL SEMANTICS OF CORE DALI

7.1 Syntax

Figure 1 defines the various syntactic categories used in specifying Core DALI, including expressions E , ground values B , values V , and contexts C .

Expressions in Core DALI include the lambda calculus with naturals. Further, the language incorporates datatypes (not necessarily just first-order), in addition to the following specialized mechanisms:

- Object-level variables $\#z$ and binders $(\#z \Rightarrow e)$,
- Pattern matching over object-bindings $\lambda(\#z \Rightarrow x).e$.
- Equality for object-bound variables $\#z =_{\#} \#z'$
- Test of whether an expression evaluates to an object-level variable (`isOVar e`).

Values, ground values, and context are used in defining the reduction semantics.

7.2 Core DALI vs. Example Language

The Core DALI has two (more primitive) forms of pattern matching than the language used in the examples: one for tagged values, one for object-level bindings. A third form of pattern matching (for object-level variables) can be easily encoded using `isOVar`.

Nested patterns are not allowed, nor are more complicated higher-order patterns directly supported: each constructor has one argument, and each higher-order pattern variable has exactly one possible free object variable in it. These simplifications make the formal development of Core DALI more manageable, without losing generality: programs in a more familiar language of our examples can be translated into equivalent, albeit more verbose Core DALI expressions.

7.3 Big Step Semantics (λ^D)

Figure 2 defines the call-by-value (CBV) big-step semantics for Core DALI. Note that this semantics does not require a *gensym* function or any freshness conditions on variables: All necessary variable renaming is handled by two standard notions of substitution [2], one for object-level variables ($\#z \in \mathbb{Z}$) and one for meta-level variables ($x \in \mathbb{X}$).

7.4 Reduction Semantics (λ^d)

Figure 1 defines the reduction semantics for Core DALI.

8. SUMMARY OF TECHNICAL DEVELOPMENT

The main technical result of our work to date is establishing the confluence property for the reduction semantics described above, and establishing (the rather non-trivial connection) between the reduction semantics and big-step semantics. In doing so, we have following closely Taha's development for the (substantially smaller) language $\lambda - U$

Syntax:

$x \in \mathbb{X}$	Normal variables	$:=$	Infinite set of names
$z \in \mathbb{Z}$	Object variables	$:=$	Infinite set of names
$f \in \mathbb{F}$	Tags	$:=$	Infinite set of names containing True and False
$F \subset \mathbb{F}$	Tag sets	$=$	Finite subsets of \mathbb{F}
$e \in \mathbb{E}$	Expressions	$:=$	$() \mid x \mid \lambda x.e \mid e \mid (e, e) \mid \pi_1 e \mid \pi_2 e \mid f e \mid \lambda^{f \in F} (f x_f).e_f \mid \#z \mid \#z \Rightarrow e \mid \lambda(\#z \Rightarrow x).e \mid \text{isOVar } e \mid e =_{\#} e$
$C \in \mathbb{C}$	Contexts	$:=$	$\square \mid \lambda x.C \mid C e \mid e C \mid (e, C) \mid (C, e) \mid \pi_1 C \mid \pi_2 C \mid \lambda^{f \in F - \{f'\}} ((f, x_i).e_i) + (f' x).C \mid f C \mid (\#z \Rightarrow C) \mid \lambda(\#z \Rightarrow x).C \mid \text{isOVar } C \mid C =_{\#} e \mid e =_{\#} C$
$b \in \mathbb{B}$	Ground Values	$:=$	$() \mid (b, b) \mid f b \mid \#z \mid \#z \Rightarrow b$
$v \in \mathbb{V}$	Values	$:=$	$() \mid \lambda x.e \mid (v, v) \mid f v \mid \lambda^{f \in F} f x_f.e_f \mid \#z \mid \#z \Rightarrow v \mid \lambda(\#z \Rightarrow x).e$
$\rho \in \mathbb{R}$	Reductions	$:=$	$\beta_1 \mid \pi_1 \mid \pi_2 \mid \beta_2 \mid \beta_3 \mid \# \mid \delta_{\text{isOVar}}$

Notions of Reduction:

$(\lambda x.e) v$	\rightarrow_{β_1}	$e[x := v]$
$\pi_1 (v_1, v_2)$	\rightarrow_{π_1}	v_1
$\pi_2 (v_1, v_2)$	\rightarrow_{π_2}	v_2
$(\lambda^{f \in F \cup \{k\}} (f x_f).e_f) (k v)$	\rightarrow_{β_2}	$e_k[x_k := v]$
$(\lambda(\#z' \Rightarrow x).e) (\#z \Rightarrow b)$	\rightarrow_{β_3}	$e[x := \lambda y.(b[\#z := y])]$
$\#z =_{\#} \#z$	$\rightarrow_{\#}$	$\text{True}()$
$\#z_1 =_{\#} \#z_2$	$\rightarrow_{\#}$	$\text{False}() \text{ if } z_1 \neq z_2$
$\text{isOVar } \#z$	$\rightarrow_{\text{isOVar}}$	$\text{True}()$
$\text{isOVar } v$	$\rightarrow_{\text{isOVar}}$	$\text{False}() \text{ if } v \neq \#z$

Reduction Semantics:

$$\frac{e_1 \rightarrow_{\rho} e_2}{C[e_1] \rightarrow C[e_2]} \rho \in \mathbb{R} \quad \frac{}{e \rightarrow^* e} \quad \frac{e_1 \rightarrow e_2 \quad e_2 \rightarrow^* e_3}{e_1 \rightarrow^* e_3}$$

Figure 1: Syntax and Reduction Semantics (λ^d) of Core DALI

[45; 47]. Taha's development is based on Takahashi parallel reduction and complete development methods for proving confluence [52], and Plotkin's "standardization" technique for showing that reductions preserve observational equivalence.

This section summarizes our technical development and states our main result, and *explains how they were useful to us in the process of designing the semantics for DALI*. The full details cannot be included in this paper, and are presented instead in a technical report available on-line [36, 40 pages].

8.1 Confluence

The first result is confluence:

THEOREM 1 (λ^d IS CONFLUENT). $\forall e_1, e_2, e \in \mathbb{E}$.

$$e_1^* \leftarrow e \rightarrow^* e_2 \implies (\exists e' \in \mathbb{E}. e_1 \rightarrow^* e' \leftarrow e_2)$$

First, we are not aware of a similar proof for a language with datatypes. Furthermore, this result establishes the existence of a confluent calculus for a language with notion of object-level binders, and analysis on these terms. In particular, this result means that DALI also provides a solution to the problem of introducing intensional analysis to MetaML in a "coherent" manner [47].

8.1.1 Role in Design of DALI

In addition to its technical role in arriving at our next result, establishing the confluence property played an important role in our design process: It drew our attention to the need for introducing the notion of ground-values, thereby prohibiting any useful mixing of object-binder and function spaces in datatypes.

In particular, analysis over object-level binders (β_3 reduction) without the restriction of the argument to ground values breaks the confluence, as is illustrated in the following example:

Suppose the notion of reduction \rightarrow_{β_3} (Figure 1) were defined as follows (we emphasize the part different from the standard definition by placing it into a box):

$$(\lambda(\#z' \Rightarrow x).e) (\#z \Rightarrow v) \rightarrow_{\beta_3} e[x := \lambda y.v[\#z := y]]$$

Now, consider the function $f \equiv (\lambda(\#w \Rightarrow x).x (\lambda u.u))$. This function takes an object-level binding as its argument and returns the body of the binding in which the object-bound variable has been replaced with the identity function $\lambda u.u$. For the application of f to the object binding $(\#z \Rightarrow (\lambda y.\#z = \#z))$, there are two possible reduction sequences:

$$\begin{aligned} f (\#z \Rightarrow (\lambda y.\#z = \#z)) &\rightarrow_{\beta_3} \lambda y.(\lambda u.u) = (\lambda u.u) \\ &\text{and} \\ f (\#z \Rightarrow (\lambda y.\#z = \#z)) &\rightarrow_{\#} f (\#z \Rightarrow (\lambda y.\text{True}())) \\ &\rightarrow_{\beta_3} \lambda y.\text{True}() \end{aligned}$$

Clearly, neither $\lambda y.\text{True}()$, nor $\lambda y.(\lambda u.u) = (\lambda u.u)$ can be

$$\begin{array}{c}
\frac{}{() \hookrightarrow ()} \quad \frac{}{\lambda x.e \hookrightarrow \lambda x.e} \quad \frac{e_1 \hookrightarrow \lambda x.e \quad e_2 \hookrightarrow e_3}{e[x := e_3] \hookrightarrow e_4} \quad \frac{e_1 \hookrightarrow \lambda^{f \in F \cup \{k\}}(f x_f).e_f \quad e_2 \hookrightarrow k e_4}{e_k[x := e_4] \hookrightarrow e_5} \quad \frac{e_1 \hookrightarrow \lambda(\#z' \Rightarrow x).e \quad e_2 \hookrightarrow \#z \Rightarrow b_3}{e[x := \lambda x'.(b_3[\#z := x'])] \hookrightarrow e_4} \\
\frac{}{e_1 e_2 \hookrightarrow e_4} \quad \frac{}{e_1 e_2 \hookrightarrow e_5} \quad \frac{}{e_1 e_2 \hookrightarrow e_4} \\
\\
\frac{e_1 \hookrightarrow e_3 \quad e_2 \hookrightarrow e_4}{(e_1, e_2) \hookrightarrow (e_3, e_4)} \quad \frac{e \hookrightarrow (e_3, e_4)}{\pi_1 e \hookrightarrow e_3} \quad \frac{e \hookrightarrow (e_3, e_4)}{\pi_2 e \hookrightarrow e_4} \quad \frac{e_1 \hookrightarrow e_2}{f_k e_1 \hookrightarrow f_k e_2} \quad \frac{}{\lambda^{f \in F} f x_f.e_f \hookrightarrow \lambda^{i \in F} f x_f.e_f} \\
\\
\frac{}{\#z \hookrightarrow \#z} \quad \frac{e_1 \hookrightarrow e_2}{\#z \Rightarrow e_1 \hookrightarrow \#z \Rightarrow e_2} \quad \frac{}{\lambda(z \Rightarrow x).e \hookrightarrow \lambda(z \Rightarrow x).e} \quad \frac{e_1 \hookrightarrow \#z \quad e_2 \hookrightarrow \#z}{e_1 =_{\#} e_2 \hookrightarrow \text{True}()} \quad \frac{e_1 \hookrightarrow \#z_1 \quad e_2 \hookrightarrow \#z_2 \quad z_1 \neq z_2}{e_1 =_{\#} e_2 \hookrightarrow \text{False}()} \\
\\
\frac{e \hookrightarrow \#z}{\text{isOVar } e \hookrightarrow \text{True}()} \quad \frac{e \hookrightarrow v \quad v \neq \#z}{\text{isOVar } e \hookrightarrow \text{False}()}
\end{array}$$

Figure 2: Big-Step Semantics (λ^D) of Core DALI

further reduced by λ^d to a common reduct: a clear counterexample for confluence.

Finally, note that the breakdown of confluence here provides a concrete illustration of one of the wide range of difficulties that can arise from mixing function spaces with “syntax”. Other examples, such as the discussion of “covers” in the context of MetaML implementation [45] require much more infrastructure to present.

8.2 Soundness

We will consider two programs to be equivalent when they can be interchanged in any context without affecting the termination (or non-termination) of the full term in which they occur. This is known as observational (or contextual) equivalence, and is defined as follows:

DEFINITION 2 (OBSERVATIONAL EQUIVALENCE). *We write $e_1 \approx e_2$ if and only if*

$$\forall C \in \mathbb{C}. (\exists v \in \mathbb{V}. C[e_1] \hookrightarrow v) \Leftrightarrow (\exists v \in \mathbb{V}. C[e_2] \hookrightarrow v)$$

Our soundness result can now be stated as simply:

THEOREM 3 (SOUNDNESS).

$$\forall e_1, e_2 \in E. e_1 \longrightarrow e_2 \implies e_1 \approx e_2$$

First, our proof for this theorem is the first operational account known to us where the soundness of such reductions for an untyped CBV functional language with datatypes is established (Using bisimilarity techniques, Pitts does present a similar result, but for a typed CBN language supporting binary sum types [41].)

Second, the soundness of these results establishes that extending the lambda calculus plus datatypes with DALI’s constructs for introducing and analyzing object-level binders and free variables at runtime *does not* injure the notion of observational equivalence in a devastating way. Certainly, it may very well be that introducing the new constructs allows us to distinguish between more terms in the language (as does introducing exceptions, for example), and this is a question for future work.

8.2.1 Role in Design of DALI

The immediate technical benefit of this result is providing technical justification for using the reductions as semantics-preserving optimizations in an implementation. But there are other benefits that we are interested in from the point of view of language design:

1. It provides us with a basic understanding of the notion of observational equivalence. In particular, in the case of this language (as is in the case for many deterministic languages), one arrives at a simple equational theory simply by changing reduction arrows into “convertibility” equalities.
2. Taha’s development [45; 47] emphasizes *partitioning* expressions into values, workables, and stucks, and establishing “monotonicity properties” from which, for example, Wright and Felleissen’s “Uniform Evaluation” [53] follows. Thus, not only do we provide the basis for posing the question of “what is a type system for datatypes with binder”, we already provide some of the technical properties needed in establishing type safety for *any type system* that we may wish to investigate.
3. Attaining this result involves constructing a number of variations of the operational semantics, and relating them formally. This process provides a substantial amount of cross-checking between various definitions, and gives a very accurate operational understanding of the kind of invariants that a type system will be expected to guarantee.

9. RELATED WORK

DALI is a functional meta-programming language, and is related as such, to many other meta-systems.

Meta-systems built with a functional programming base include MetaML [47; 51], λ^\square [12] and λ° [11]. These differ from DALI in that they are homogeneous systems, where the meta- and object-languages are the same. None of these systems provide mechanisms for analyzing the structure of object-programs.

Theorem prover based meta-systems have been constructed for several kinds of logics. Implementations of classical logics include the HOL [18] theorem prover, Isabelle [35], and the Prototype Verification System (PVS) [34]. Implementations of constructive (or intuitionistic) logics include Elf [37], Coq [10; 3], Nuprl [9] and Lego [42].

Finally, there are logic programming languages with meta-programming extensions, λ -Prolog [31; 15; 27], and L_λ [24]. These are prolog-like languages with extensions for representing and analyzing object-programs whose representations are based on the λ -calculus.

Of these systems, Isabelle, Elf, λ -Prolog, and L_λ use some sort of higher-order abstract syntax to represent object terms. Of these, all but L_λ , use higher order unification to implement intensional analysis of object terms. Higher order unification is in general undecidable, and does not guarantee a most general unifier.

L_λ implements a subset of lambda-Prolog, where intensional analysis is syntactically restricted to a form which is decidable using unification on higher-order patterns. It is this idea transferred to the functional programming world that is the basis for ML_λ and DALI.

The term *higher-order abstract syntax* was originated by Pfenning and Elliott [38]. This work provided a basis for automating reasoning in LF[19], and was used as the basis for the implementation of Pfenning's Elf[37] and its successor Twelf[40].

9.1 DALI vs. ML_λ

Dale Miller [23] describes ML_λ , a proposal to extend ML to handle bound variables in data-types. The idea of representing object-level bindings, in a functional language, using a binding construct different from the function abstraction of the meta-language derives from this paper. While our work takes Miller's proposed extensions as its basis, there are some differences:

- We distill the main ideas of Miller's ML_λ into a basic calculus of core DALI. We concentrate on the reduction semantics and equational theories of such a language. To the authors' knowledge, this work is the first instance of a sound reduction semantics for a functional language supporting binding constructs in data-types.
- We abandon the notion of *function extension* that allows extending the domain of arbitrary ML functions within the scope of an object-level variable. We find function extension needlessly difficult to model in a reduction system, and seek to introduce an alternative construct: patterns that match object-level variables. We conjecture that, together with equality over object-level variables, one can circumvent function extension without loss of expressiveness or good programming style.
- We abandon the notion of object-level application [23]. Rather, pattern matching on object-level bindings binds higher-order pattern variables to functions that perform appropriate substitutions directly, thus further simplifying formal development, and, in practical terms,

internalizing object-level variable substitution, which in [23] must be defined separately for each data-type.

However, internalization of such object-level substitution in presence of extensional function values is not without cost: we had to resort to a fine distinction between ground (or equality) values and the more standard notion of values in such calculi, and adjust evaluation and reduction to restrict the analysis of object-language terms to preserve soundness and confluence of the calculus.

DALI differs from most of the other work discussed above in following ways:

- It is functional and deterministic, and is presented as an extension of a standard CBV functional language. It provides support for higher-order syntax by providing a small number of new language constructs.
- The formal properties we have proven about the language suggest that the new features integrate well with the host functional language.
- The reduction semantics we provide gives rise to a simple equational theory that can be used to reason about program equivalence.

10. CONCLUSIONS AND FUTURE WORK

In this paper we have shown that a functional programming language with support for higher order abstract syntax through an additional object-level binding construct can be assigned a simple big-step semantics. We have defined a reduction semantics and presented important results of confluence and soundness w.r.t. evaluation of this reduction semantics for DALI. After this initial success much work remains to be done. In particular:

- Developing a basic type system for DALI. In addition to the traditional notions of safety there are some efficiency concerns that we expect that a type system could be used to alleviate. In particular, the discharge operation and the use of the ground-value restriction b in the semantics would incur significant run-time penalties in an implementation. We expect that an appropriate type system could help avoid these.
- Integrating with multi-stage programming. In particular, DALI meta-programming utility is orthogonal to that of multi-stage programming [49; 48; 48; 29; 4]: with DALI, the object language is allowed to vary, and intensional analysis is supported. Note, however, that DALI does support the hygienic synthesis of object code, although in a manner less concise than those of multi-stage programming languages. Finally, whereas it has been demonstrated that the former can guarantee that the synthesized code is type correct, the only guarantee that we have at the moment with DALI is that the synthesized code is syntactically correct.
- An implementation of a full programming language environment based on DALI. Although a full implementation of DALI is missing at the moment, the mechanisms of higher-order pattern matching and analysis

of object-level bindings has been implemented by Tim Sheard as an experimental feature of the MetaML interpreter [43].

From the point of view of semantic language design, in reproducing Taha's technical development of MetaML, we have found that all the proofs could be carried out in a systematic manner for the (considerably larger) language at hand, and that many of the proofs remain literally unchanged. This seems to be primarily due to the use of the notion of "workables" in parameterizing the various lemmata. In future work, we intend to investigate the extent to which this development can be generalized.

11. REFERENCES

- [1] H. Abramson and M. Rogers. *Meta-Programming in Logic Programming*. 1989.
- [2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.
- [3] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.
- [4] Zine El-Abidine Benaïssa, Eugenio Moggi, Walid Taha, and Tim Sheard. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)*, July 1999. To appear.
- [5] Nikolaj Björner. Type checking meta programs. In *Workshop on Logical Frameworks and Meta-language*, Paris, September 1999.
- [6] R. M. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In *Proceedings of Advanced Course on Abstract Software Specifications*, pages 292–332. Springer-Verlag, Lecture Notes in Computer Science, 1980.
- [7] Rod Burstall and Joseph Goguen. An informal introduction to specifications using Clear. In Robert Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.
- [8] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [9] Robert L. Constable, S. Allen, h. Bromely, W. Cleveland, et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.
- [10] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.
- [11] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, July 1996. IEEE Computer Society Press.
- [12] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 258–270, St. Petersburg Beach, January 1996.
- [13] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.
- [14] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan 1996*, pages 284–294. ACM Press, New York, 1996.
- [15] Amy Felty, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. Lambda prolog: An extended logic programming language. In E. Lusk; R. Overbeek, editor, *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 754–755, Berlin, May 1988. Springer.
- [16] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 193–202, Trento, Italy, July 1999. IEEE Computer Society Press.
- [17] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.
- [18] M. J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [19] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings Symposium on Logic in Computer Science*, pages 194–204, Washington, June 1987. IEEE Computer Society Press. The conference was held at Cornell University, Ithaca, New York.
- [20] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, Trento, Italy, July 1999. IEEE Computer Society Press.
- [21] Gérard Huet and B. Lang. Proving and applying program transformations expressed with second-order logic. *Acta Informatica*, 11:31–55, 1978.
- [22] Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993. Available online from <http://www.dina.dk/~sestoft>.
- [23] Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Informal Proceedings of the Logical Frameworks BRA Workshop*, June 1990. Available as UPenn CIS technical report MS-CIS-90-59.
- [24] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1, 1991.
- [25] Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in *Lecture Notes in Artificial Intelligence*, pages 322–337. Springer-Verlag, 1992.
- [26] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, October 1992.
- [27] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

- [28] E. Moggi. Functor categories and two-level languages. In *FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
- [29] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.
- [30] Eugenio Moggi. A categorical account of two-level languages. In *Mathematics Foundations of Program Semantics*. Elsevier Science, 1997.
- [31] Gopalan Nadathur and Dale Miller. An overview of lambda prolog. Technical Report DUKE-TR-1988-14, Duke University, January 1, 1988.
- [32] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.
- [33] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000, USA. Available online from <ftp://cse.ogi.edu/pub/tech-reports/README.html>. Last viewed August 1999.
- [34] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. *Lecture Notes in Computer Science*, 607:748–??, 1992.
- [35] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
- [36] Emir Pašalić, Tim Sheard, and Walid Taha. DALI: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Technical Report CSE-00-007, OGI, March 2000. Available from [33].
- [37] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.
- [38] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, June 1988.
- [39] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. Ergo Report 88-065, Carnegie Mellon University, Pittsburgh, July 1988.
- [40] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10, 1999. Springer-Verlag.
- [41] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1995. Based on lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge UK, September 1995.
- [42] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
- [43] Tim Sheard and Zino Benaissa. MetaML v1.0. <http://www.cse.ogi.edu/PacSoft/projects/metaml>, March 2000.
- [44] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, January 1998.
- [45] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Revised October 99. Available from author (taha@cs.chalmers.se).
- [46] Walid Taha. Runtime tag elimination (extended abstract). Submitted for publication. Available online, January 2000.
- [47] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *2000 SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, January 2000.
- [48] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, July 1998.
- [49] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997. An extended and revised version appears in [51].
- [50] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. Technical Report CSE-99-007, Department of Computer Science, Oregon Graduate Institute, January 1999. Extended version of [49]. Available from [33].
- [51] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000. In Press. Revised version of [50].
- [52] Masako Takahashi. Parallel reductions in λ -calculus. *Information and Computation*, 118(1):120–127, April 1995.
- [53] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

DALI: An Untyped CBV Operational Semantics and Equational Theory for Datatypes with Binders (Technical Development)

Emir Pašalić, Tim Sheard*, Walid Taha**

Oregon Graduate Institute *and* Chalmers University of Technology

Abstract. This report presents the basic definitions and formal development of DALI, a novel extension of the CBV lambda calculus. DALI is based on a proposal by Miller, and provides an elegant and well-behaved notion of object-variables. The notion is *elegant* because it avoids any explicit need for a “gensym” or “newname” operation that is often needed in such systems. The notion is *well-behaved* in that it induces a computationally adequate equational theory.

Our development follows the one employed in Taha’s thesis [4] for MetaML[7, 5]. The development was easy to adapt, and some of the major lemmas and proofs remained essentially unchanged. This success is further evidence of the robustness of both the observations of Takahashi on the notion of parallel reduction and the original “standardisation” development by Plotkin for the CBV and CBN lambda calculus.

* The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-96-C-0161, NSF Grants CCR-9803880 and IRI-9625462, and the Department of Defense.

** Supported by a Postdoctoral Fellowship, funded by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403.

Table of Contents

DALI: An Untyped CBV Operational Semantics and Equational Theory for Datatypes with Binders (Technical Development)	1
<i>Emir Pašalić, Tim Sheard, Walid Taha</i>	
1 Preliminaries	4
1.1 A Note Regarding the First Revision	4
1.2 Notes on the Technical Development	4
2 Definitions	5
2.1 $\mathbb{X}, \mathbb{Z}, \mathbb{F}, \mathbb{E}, \mathbb{C}, \mathbb{V}, \mathbb{B}, \mathbb{W}, \mathbb{S}$ Set Definitions	5
2.2 $[-]: \mathbb{C} \times \mathbb{E} \rightarrow \mathbb{E}$ Context filling	6
2.3 $[- := -]: \mathbb{E} \times \mathbb{X} \times \mathbb{E} \rightarrow \mathbb{E}$ and $[-\# := -]: \mathbb{B} \times \mathbb{Z} \times \mathbb{X} \rightarrow \mathbb{E}$ Substitution	6
2.4 $[- \rightarrow -]: \mathbb{E} \times \mathbb{R} \times \mathbb{E} \rightarrow \mathbb{E}$ Notions of Reduction	7
2.5 $[- \rightarrow -] \subseteq \mathbb{E} \times \mathbb{E}$ and $[- \rightarrow^* -] \subseteq \mathbb{E} \times \mathbb{E}$ Compatible Reduction and the Reduction Relation	7
2.6 Notation	7
2.7 $[- \gg -] \subseteq \mathbb{E} \times \mathbb{E}$ Parallel Reduction	7
2.8 $[- \overset{M}{\gg} -]: \mathbb{E} \rightarrow \mathbb{E}$ Parallel Reduction with Complexity	8
2.9 $[-!]: \mathbb{E} \rightarrow \mathbb{E}$ Complete Development	8
2.10 $[- \hookrightarrow -]: \mathbb{E} \rightarrow \mathbb{E}$ Big-Step Semantics	9
2.11 $[- \mapsto -]: \mathbb{E} \rightarrow \mathbb{E}$ Left Reduction	9
3 Basic Properties	9
3.1 Substitution	9
3.2 Parallel Reduction	10
3.3 Substitution Lemma for Parallel Reduction with Complexity	14
3.4 Big-Step Semantics	16
3.5 Classes	16
3.6 Parallel Reduction and Classes	17
3.7 Left Reduction	19
4 General Part of Confluence	21
4.1 Parallel Reduction is Diamond	21
4.2 Takahashi's Property	21
4.3 Main Confluence Result	21
5 Special Part of Confluence	21
5.1 Takahashi's Property	21
6 General Part of Computational Adequacy of Reduction Semantics	23
6.1 Main Soundness Theorem	23
6.2 Reduction is in Evaluation	24
6.3 Push Back	26
7 Special Part of Computational Adequacy of Reduction Semantics	26
7.1 Evaluation is in Reduction	27
7.2 Left Reduction is in Reduction	28
7.3 Left Reduction is Evaluation	29
7.4 Transition Lemma	31
7.5 Permutation Lemma	33

8	Acknowledgements	35
---	------------------------	----

1 Preliminaries

1.1 A Note Regarding the First Revision

This document presents the technical development of a core calculus of DALI, a CBV language which provides support for datatypes with bindings. The proofs are given in as much detail as the space would allow it.

Due to the time constraints, many typographical and stylistic changes we wished to make in order to make the document more readable and compact are still missing. We intend to address these problems regarding our presentation in a forthcoming revision. e.p.

The document is organized as follows: Section 2 presents definitions of the various notions that will be used throughout the proof. Section 3 proves various basic properties of those notions. Section 4 provides the general proof of confluence, taken directly from [5]. Section 5 proves necessary lemmas for DALI that are required for the general argument in section 4. Section 6 gives a general argument for soundness of DALI, adapted directly from [5]. Section 7 provides proofs for DALI-specific lemmas required in Section 6.

1.2 Notes on the Technical Development

The development proceeds in a number of steps. Some of these steps introduce auxiliary constructs that are useful in constructing the proofs of properties in which we are interested. These constructs include well-known notions, such as *complete development* and parallel reduction, which are recast in the setting of DALI.

Each new notion usually has certain basic properties associated with it, about which we prove lemmas (Section 3). These basic properties, beside being useful in latter proofs, also ensure that the definitions of various notions are sensible and correspond to their equivalents in more well-known calculi.

We introduce a big-step semantics (or evaluation function where the order of evaluation of subterms is deterministic), and a reduction semantics (where the order in which the rules are applied is immaterial).

The main result of this paper is that reduction preserves evaluation. That is, applying any sequence of reduction rules to a term does not change the value to which it evaluates. Thus the reduction calculus can be used to transform one program into another program with equivalent meaning.

Our development includes the following steps:

- We define a core language (DALI, Section 2) that exhibits only the essential features and properties we wish to develop theoretically.
- We define a big-step semantics (\hookrightarrow), a deterministic partial function that defines the notion of evaluation of DALI programs.
- We define a reduction semantics (λ^d), based on primitive notions of reduction (e.g., β), lifted into arbitrary contexts to obtain a compatible reduction relation (\longrightarrow), as well as its reflexive transitive closure (\longrightarrow^*).
- We define *parallel reduction* (\gg), for DALI, a relation between terms that allows multiple reductions to be performed at the same time, nondeterministically.
- We define *left reduction* (\mapsto), a function on terms that performs the work of big-step semantics in a number of small steps.
- We prove important properties of these constructs:
 1. λ^d is equivalent to parallel reduction. This allows us to replace reduction semantics with parallel reductions in our proofs, since parallel reduction is considerably less difficult to reason about than λ^d .
 2. We prove equivalence of the transitive closure of left reduction (\mapsto^*) and big-step evaluation. Similarly to parallel reduction, this allows us to reason about \mapsto^* instead of \hookrightarrow in our proofs which simplifies the development.
- The first major result of our work on DALI is the proof of confluence of λ^d . (Sections 4 and 5) We prove that parallel reduction is confluent, which by equivalence of the transitive closures of λ^d and parallel reduction allows us to conclude that λ^d is confluent as well.
- Based on Taha's soundness proof of MetaML [5], the partiality of left reduction function induces a partitioning of the set of expressions, called expression classes. The three term classes are inductively defined and are called *values*, *workables*, and *stucks*.
 1. *Workables* are expressions on which left reduction is defined, i.e., can be advanced by left reduction. In other words, workables may left-reduce, in one or more steps, to either values, stucks or other workables.

2. *Values* are expressions to which workables may left-reduce in one or more steps but on which left reduction cannot proceed further. They correspond exactly to the set of values defined in Section 2.
 3. *Stucks* are expressions that cannot be advanced by left reduction, but are not values.
- A further development is to prove certain monotonic properties of parallel reduction with respect to term classes. In particular, values and stucks can be only further reduced to other values or stucks respectively, and most importantly, if a result of the parallel reduction is a value, it must have been obtained either from another value, or from a workable.
 - Finally, the main result of this paper, the soundness of reduction semantics, is reduced to proving two goals
 1. A (terminating) evaluation of a term to a value implies a finite sequence of λ^d steps in which the original term reduces to the same value.
 2. A finite sequence of reductions of a term e to a value v implies that there exists a value v' to which the original term evaluates, and which can be then reduced in some finite number of steps to the original value v .

The first goal is not difficult to prove by straightforward induction. The second goal, however, requires several transformations:

1. Using the equivalence of evaluation and left reduction's reflexive transitive closure, and the equivalence of λ^d and parallel reduction, the goal is restated in terms of left reduction and parallel reduction.
2. Three important lemmas are used in proving this goal: *push-back*, *transition* and *permutation*. These lemmas allow rearranging of the order of left and parallel reductions in finite reduction chains until the goal is reached.

The formulation of the soundness proof technique is most directly due to Taha's work on MetaML[4, 5]. In proving soundness of DALI, we were able to directly reuse both the structure and certain key lemmas of this proof. For others, it was necessary to re-prove them in the context of the new language, but their basic formulation remained unchanged.

2 Definitions

2.1 $\mathbf{X, Z, F, E, C, V, B, W, S}$ Set Definitions

We will use a **BNF-like** notation for specifying a number of inductively defined sets that will be used throughout this report. Two non-standard notations will be used for conciseness:

- **Lambda terms with patterns** are written $\lambda^{f \in \{f_1, \dots, f_n\}}(f \ x_f).e_f$, meaning, a sequence of simple lambda abstractions $\lambda x_1.e_1, \dots, \lambda x_n.e_n$ corresponding to branches of a lambda abstraction indexed by the various tags f_1, \dots, f_n .
- **Set difference** is written $\mathbb{V} \setminus v$, meaning, all possible elements of \mathbb{V} except the ones matching the pattern particular pattern v . For example, if \mathbb{N} is the set of naturals, then $n+1$ is a pattern that matches naturals greater than zero, and $\mathbb{N} \setminus (n+1)$ is simply the natural number zero. This notation allows us to avoid having a definition of stuck terms \mathbb{S} that is quadratic in the number of constructs in expressions \mathbb{E} .

We will make use of **Barendregt's variable convention**, and state the set of bound and free variables in expressions occurring in any formula or statement should be taken by the reader to be distinct.

Definition 1

$x \in \mathbb{X}$ Normal variables	$:=$ Infinite set of names
$z \in \mathbb{Z}$ Object variables	$:=$ Infinite set of names
$f \in \mathbb{F}$ Tags	$:=$ Infinite set of names containing True and False
$F \subset \mathbb{F}$ Tag sets	$=$ Finite subsets of \mathbb{F}
$e \in \mathbb{E}$ Expressions	$:= () \mid x \mid \lambda x.e \mid e e \mid (e, e) \mid \pi_1 e \mid \pi_2 e \mid f e \mid \lambda^{f \in F} (f x_f).e_f \mid$ $\#z \mid \#z \Rightarrow e \mid \lambda(\#z \Rightarrow x).e \mid \text{isOVar } e \mid e =_{\#} e$
$C \in \mathbb{C}$ Contexts	$:= [] \mid \lambda x.C \mid C e \mid e C \mid (e, C) \mid (C, e) \mid \pi_1 C \mid \pi_2 C \mid$ $\lambda^{f \in F - \{f'\}} ((f_i x_i).e_i) + (f' x).C \mid f C \mid (\#z \Rightarrow C) \mid \lambda(\#z \Rightarrow x).C$ $\mid \text{isOVar } C \mid C =_{\#} e \mid e =_{\#} C$
$b \in \mathbb{B}$ Based Values	$:= () \mid (b, b) \mid f b \mid \#z \mid \#z \Rightarrow b$
$v \in \mathbb{V}$ Values	$:= () \mid \lambda x.e \mid (v, v) \mid f v \mid \lambda^{f \in F} f x_f.e_f \mid \#z \mid \#z \Rightarrow v \mid \lambda(\#z \Rightarrow x).e$
$w \in \mathbb{W}$ Workables	$:= (\lambda x.e) v \mid (\lambda(\#z \Rightarrow x).e) (\#z \Rightarrow b) \mid w e \mid v w \mid f w \mid \#z \Rightarrow w \mid$ $(w, e) \mid (v, w) \mid \pi_n w \mid \pi_n(v, v)$ $\mid w = e \mid \#z = w \mid \#z = \#z'$ $\mid \text{isOVar } v \mid \text{isOVar } w$
$s \in \mathbb{S}$ Stucks	$:= x \mid s e \mid v s \mid (s, e) \mid (v, s) \mid \pi_n s \mid \pi_n (\mathbb{V} \setminus (v, v)) \mid f s$ $\mid (\lambda^{f \in F} f x_f.e_f) (\mathbb{V} \setminus f v) \mid (\#z \Rightarrow s)$ $\mid (\mathbb{V} \setminus \#z) = e \mid \#z = (\mathbb{V} \setminus \#z) \mid s = e \mid \#z = s$ $\mid (\lambda(\#z \Rightarrow x).e) (\mathbb{V} \setminus (\#z \Rightarrow b))$ $\mid (\mathbb{V} \setminus \lambda^e) v \text{ where } \lambda^e = \lambda x.e + \lambda^{f \in F} f x_f.e_f + \lambda(\#z \Rightarrow x).e$ $\mid \text{isOVar } s$
$\rho \in \mathbb{R}$ Reductions	$:= \beta_1 \mid \pi_1 \mid \pi_2 \mid \beta_2 \mid \beta_3 \mid \# \mid \delta_{\text{isOVar}}$

2.2 $\boxed{[-]: \mathbb{C} \times \mathbb{E} \rightarrow \mathbb{E}}$ Context filling

Definition 2

$$\begin{aligned}
[] [e'] &= e' & (\lambda x.C) [e'] &= \lambda x.(C [e']) & (\lambda \#z \Rightarrow x.C) [e'] &= (\lambda \#z \Rightarrow x.C [e']) & (C e) [e'] &= C [e'] e & (e C) [e'] &= e C [e'] \\
(\#z \Rightarrow C) [e'] &= (\#z \Rightarrow C [e']) & (C =_{\#} e) [e'] &= C [e'] = e & (e =_{\#} C) [e'] &= e = C [e'] \\
(e, C) [e'] &= (e, C [e']) & (C, e) [e'] &= (C [e'], e) & (\pi_1 C) [e'] &= \pi_1 (C [e']) & (\pi_2 C) [e'] &= \pi_2 (C [e']) & (f C) [e'] &= f (C [e']) \\
(\lambda^{f \in F - \{f'\}} ((f x_i).e_i) + (f' x).C) [e'] &= (\lambda^{f \in F - \{f'\}} ((f x_i).e_i) + (f' x).C [e']) \\
(\text{isOVar } C) [e'] &= \text{isOVar } C [e']
\end{aligned}$$

2.3 $\boxed{[-] := [-]: \mathbb{E} \times \mathbb{X} \times \mathbb{E} \rightarrow \mathbb{E}}$ and $\boxed{[-] \# [-] := [-]: \mathbb{B} \times \mathbb{Z} \times \mathbb{X} \rightarrow \mathbb{E}}$ Substitution

Definition 3

$$\begin{aligned}
() [x := e_3] &= () \\
x [x := e_3] &= e_3 \\
x' [x := e_3] &= x', & x \neq x' \\
\lambda x'.e [x := e_3] &= \lambda x'.(e [x := e_3]) \\
e_1 e_2 [x := e_3] &= (e_1 [x := e_3]) (e_2 [x := e_3]) \\
(e_1, e_2) [x := e_3] &= (e_1 [x := e_3], e_2 [x := e_3]) \\
\pi_1 e [x := e_3] &= \pi_1 (e [x := e_3]) \\
\pi_2 e [x := e_3] &= \pi_2 (e [x := e_3]) \\
f e_1 [x := e_3] &= f (e_1 [x := e_3]) \\
\lambda^{f \in F} (f x_f).e_f [x := e_3] &= \lambda^{f \in F} (f x'_f).(e_f [x := e_3]) \\
\#z [x := e_3] &= \#z \\
\#z' \Rightarrow e [x := e_3] &= \#z' \Rightarrow (e [x := e_3]) \\
\lambda(\#z \Rightarrow x').e [x := e_3] &= \lambda(\#z \Rightarrow x').(e [x := e_3]) \\
e_1 =_{\#} e_2 [x := e_3] &= (e_1 [x := e_3]) =_{\#} (e_2 [x := e_3]) \\
\text{isOVar } e [x := e_3] &= \text{isOVar } (e [x := e_3])
\end{aligned}$$

$$\begin{aligned}
() [\#z := x] &= () \\
f b [\#z := x] &= f (b [\#z := x]) \\
\#z [\#z := x] &= x \\
\#z' [\#z := x] &= \#z', & \#z' \neq \#z \\
\#z' \Rightarrow b [\#z := x] &= z' \Rightarrow (b [\#z := x])
\end{aligned}$$

Remark 4 Note that the above definition of substitution uses Barendregt's variable convention, and thus no explicit side-conditions on renaming substitutions over binding constructs are necessary.

2.4 $\boxed{- \longrightarrow - : \mathbb{E} \times \mathbb{R} \times \mathbb{E} \rightarrow \mathbb{E}}$ Notions of Reduction

Definition 5

$$\begin{aligned}
 (\lambda x. e) v &\longrightarrow_{\beta_1} e[x := v] \\
 \pi_1(v_1, v_2) &\longrightarrow_{\pi_1} v_1 \\
 \pi_2(v_1, v_2) &\longrightarrow_{\pi_2} v_2 \\
 (\lambda^{i \in L - \{k\}} (f \ x_i). e_i) (k \ v) &\longrightarrow_{\beta_2} e_k[x_k := v] \\
 (\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) &\longrightarrow_{\beta_3} e[x := \lambda x. (b[\#z := x])] \\
 \#z = \# \ \#z &\longrightarrow_{\#} \text{True}() \\
 \#z_1 = \# \ \#z_2 &\longrightarrow_{\#} \text{False}() \text{ if } z_1 \neq z_2 \\
 \text{isOVar } \#z &\longrightarrow_{\text{isOVar}} \text{True}() \\
 \text{isOVar } v &\longrightarrow_{\text{isOVar}} \text{False}() \text{ if } v \neq \#z
 \end{aligned}$$

2.5 $\boxed{- \longrightarrow - \subset \mathbb{E} \times \mathbb{E}}$ and $\boxed{- \longrightarrow^* - \subset \mathbb{E} \times \mathbb{E}}$ Compatible Reduction and the Reduction Relation

Definition 6

$$\frac{e_1 \longrightarrow_{\rho} e_2}{C[e_1] \longrightarrow C[e_2]} \rho \in \mathbb{R} \qquad \frac{}{e \longrightarrow^* e} \quad \frac{e_1 \longrightarrow e_2 \quad e_2 \longrightarrow^* e_3}{e_1 \longrightarrow^* e_3}$$

2.6 Notation

Notation 7 (Relation Composition) For any two relations \oplus and \otimes , we write $a \oplus b \otimes c$ as a shorthand for $(a \oplus b) \wedge (b \otimes c)$.

2.7 $\boxed{- \gg - \subseteq \mathbb{E} \times \mathbb{E}}$ Parallel Reduction

In order to prove the two key lemmas presented in this section, we will need to reason by induction on the “complexity” of parallel reduction. Thus, we will use the following definition of parallel reduction with an associated complexity measure. Where complexity is not relevant, we will simply omit it to avoid unnecessary clutter.

2.8 $\boxed{- \gg^M \cdot : \mathbb{E} \rightarrow \mathbb{E}}$ Parallel Reduction with Complexity

$$\begin{array}{c}
\frac{}{() \gg^0 ()} \quad \frac{}{x \gg^0 x} \quad \frac{e_1 \gg^N e_2}{\lambda x. e_1 \gg^N \lambda x. e_2} \quad \frac{e_1 \gg^M e_3 \quad e_2 \gg^N e_4}{e_1 e_2 \gg^{M+N} e_3 e_4} \quad \frac{e_1 \gg^M e_3 \quad v_1 \gg^N v_2}{(\lambda x. e_1) v_1 \gg^{M+\#(x, e_3)N+1} e_3[x := v_2]} \\
\\
\frac{e_1 \gg^M e_2}{(\lambda(\#z \Rightarrow x). e_1) (\#z \Rightarrow b) \gg^{M+\#(x, e_2)N+1} e_2[x := \lambda x. b[\#z := x]]} \quad \frac{e_k \gg^M e'_k \quad v_1 \gg^N v_2}{(\lambda^{i \in F} f_i x_i. e_i) (f_k v_1) \gg^{M+\#(x_k, e'_k)N+1} e'_k[x_k := v_2]} \\
\\
\frac{e_1 \gg^M e_3 \quad e_2 \gg^N e_4}{(e_1, e_2) \gg^{M+N} (e_2, e_4)} \quad \frac{e_1 \gg^M e_2}{\pi_1 e_1 \gg^M \pi_1 e_2} \quad \frac{e_1 \gg^M e_2}{\pi_2 e_1 \gg^M \pi_2 e_2} \quad \frac{v_1 \gg^N v'_1}{\pi_1 (v_1, v_2) \gg^{N+1} v'_1} \quad \frac{v_2 \gg^N v'_2}{\pi_2 (v_1, v_2) \gg^{N+1} v'_2} \\
\\
\frac{e_1 \gg^M e_2}{f e_1 \gg^M f e_2} \quad \frac{e_f \gg^N e'_f}{\lambda^{f \in F} f x_f. e_f \gg^{\Sigma N_f} \lambda^{f \in F} f x_f. e'_f} \quad \frac{}{\#z \gg^0 \#z} \quad \frac{e_1 \gg^M e_2}{(\#z \Rightarrow e_1) \gg^M (\#z \Rightarrow e_2)} \quad \frac{e_1 \gg^N e_2}{\lambda(\#z \Rightarrow x). e_1 \gg^N \lambda(\#z \Rightarrow x). e_2} \\
\\
\frac{}{\#z = \# \#z \gg^1 \text{True}()} \quad \frac{\#z_1 \neq \#z_2}{\#z_1 = \# \#z_2 \gg^1 \text{False}()} \quad \frac{e_1 \gg^M e_2 \quad e_2 \gg^N e_4}{e_1 = \# e_2 \gg^{M+N} e_2 = \# e_4} \\
\\
\frac{e \gg^X e'}{\text{isOVar } e \gg^X \text{isOVar } e'} \quad \frac{v \neq \#z}{\text{isOVar } v \gg^1 \text{False}()} \\
\\
\frac{}{\text{isOVar } \#z \gg^1 \text{True}()}
\end{array}$$

where $\#(x, e)$ is the number of free occurrences of the variable x in the term e .

2.9 $\boxed{! \cdot : \mathbb{E} \rightarrow \mathbb{E}}$ Complete Development

Definition 8

$$\begin{aligned}
!x &\equiv x \\
!(\lambda x. e) &\equiv \lambda x. !e \\
!(e_1 e_2) &\equiv !e_1 !e_2 \text{ if } e_1 e_2 \neq \begin{cases} (\lambda x. e_3) v_4 \\ (\lambda^{i \in F \cup \{k\}} (f x_f. e_f)) (k v) \\ (\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) \end{cases} \\
!(\lambda x. e) v &\equiv !e[x := !v] \\
!(e_1, e_2) &\equiv (!e_1, !e_2) \\
!(\pi_n e) &\equiv \pi_n !e \quad e \neq (v_1, v_2) \\
!(\pi_1 (v_1, v_2)) &\equiv !v_1 \\
!(\pi_2 (v_1, v_2)) &\equiv !v_2 \\
!(f e) &\equiv f !e \\
!((\lambda^{f \in F \cup \{k\}} f x_f. e_f) (k v)) &\equiv !e_k[x_k := !v] \\
!\#z &\equiv \#z \\
!(\#z \Rightarrow e) &\equiv \#z \Rightarrow !e \\
!(\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) &\equiv !e[x := \lambda x'. b[\#z := x']] \\
!(e_1 = \# e_2) &\equiv !e_1 = \# !e_2 \quad \text{if } e_1, e_2 \notin \mathbb{Z} \\
!(\#z = \#z) &\equiv \text{True}() \\
!(\#z_1 = \#z_2) &\equiv \text{False}() \quad \text{if } \#z_1 \neq \#z_2 \\
!(\text{isOVar } e) &\equiv \text{isOVar } !e \quad \text{if } e \notin \mathbb{V} \\
!(\text{isOVar } \#z) &\equiv \text{True}() \\
!(\text{isOVar } v) &\equiv \text{False}() \quad \text{if } v \neq \#z
\end{aligned}$$

2.10 $\boxed{- \hookrightarrow \cdot : \mathbb{E} \rightarrow \mathbb{E}}$ Big-Step Semantics

Definition 9

$$\begin{array}{c}
\frac{}{() \hookrightarrow ()} \quad \frac{}{\lambda x.e \hookrightarrow \lambda x.e} \quad \frac{e_1 \hookrightarrow \lambda x.e \quad e_2 \hookrightarrow e_3 \quad e[x:=e_3] \hookrightarrow e_4}{e_1 e_2 \hookrightarrow e_4} \quad \frac{e_1 \hookrightarrow \lambda^{f \in F \cup \{k\}}(f x_f).e_f \quad e_2 \hookrightarrow k e_4 \quad e_k[x:=e_4] \hookrightarrow e_5}{e_1 e_2 \hookrightarrow e_5} \quad \frac{e_1 \hookrightarrow \lambda(\# _ \Rightarrow x).e \quad e_2 \hookrightarrow \#z \Rightarrow b_3 \quad e[x:=\lambda x'.(b_3[\#z:=x'])] \hookrightarrow e_4}{e_1 e_2 \hookrightarrow e_4} \\
\\
\frac{e_1 \hookrightarrow e_3 \quad e_2 \hookrightarrow e_4}{(e_1, e_2) \hookrightarrow (e_3, e_4)} \quad \frac{e \hookrightarrow (e_3, e_4)}{\pi_1 e \hookrightarrow e_3} \quad \frac{e \hookrightarrow (e_3, e_4)}{\pi_2 e \hookrightarrow e_4} \quad \frac{e_1 \hookrightarrow e_2}{f_k e_1 \hookrightarrow f_k e_2} \quad \frac{}{\lambda^{f \in F} f x_f.e_f \hookrightarrow \lambda^{i \in F} f x_f.e_f} \\
\\
\frac{}{\#z \hookrightarrow \#z} \quad \frac{e_1 \hookrightarrow e_2}{\#z \Rightarrow e_1 \hookrightarrow \#z \Rightarrow e_2} \quad \frac{}{\lambda(z \Rightarrow x).e \hookrightarrow \lambda(z \Rightarrow x).e} \quad \frac{e_1 \hookrightarrow \#z \quad e_2 \hookrightarrow \#z}{e_1 =_{\#} e_2 \hookrightarrow \text{True}()} \quad \frac{e_1 \hookrightarrow \#z_1 \quad e_2 \hookrightarrow \#z_2 \quad z_1 \neq z_2}{e_1 =_{\#} e_2 \hookrightarrow \text{False}()} \\
\\
\frac{e \hookrightarrow \#z}{\text{isOVar } e \hookrightarrow \text{True}()} \quad \frac{e \hookrightarrow v \quad v \neq \#z}{\text{isOVar } e \hookrightarrow \text{False}()}
\end{array}$$

Remark 10 Note that the use of b in the definition of the semantics of the application of a case over object-binders is an expensive runtime check. However, we expect that it should be possible to eliminate the need for this check by an appropriate type system that restricts “analysable” terms to base values.

2.11 $\boxed{- \mapsto \cdot : \mathbb{E} \rightarrow \mathbb{E}}$ Left Reduction

The notion of left reduction is intended to capture precisely the reductions performed by the big-step semantics, in a small-step manner. Note that the simplicity of the definition depends on the fact that the partial function being defined is *not* defined on values. That is, we expect that there is no e such that $v \mapsto e$.

Lemma 22 says that the set of workables characterises exactly the set of terms that can be advanced by left reduction.

Definition 11

$$\begin{array}{c}
\frac{}{(\lambda x.e) v \mapsto e[x:=v]} \quad \frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \quad \frac{e \mapsto e'}{v e_2 \mapsto v e'} \quad \frac{e_1 \mapsto e'_1}{(e_1, e_2) \mapsto (e'_1, e_2)} \quad \frac{e \mapsto e'}{(v, e) \mapsto (v, e')} \\
\\
\frac{e \mapsto e'}{\pi_n e \mapsto \pi_n e'} \quad \frac{}{\pi_1 (v_1, v_2) \mapsto v_1} \\
\\
\frac{}{\pi_2 (v_1, v_2) \mapsto v_2} \quad \frac{}{(\lambda^{f \in L \cup \{k\}}(f x_f).e_f) (k v) \mapsto e_k[x_k:=v]} \quad \frac{e \mapsto e'}{f e \mapsto f e'} \\
\\
\frac{e \mapsto e'}{\#z \Rightarrow e \mapsto \#z \Rightarrow e'} \quad \frac{}{(\lambda(\#z \Rightarrow x).e) (\#z' \Rightarrow b) \mapsto e[x:=\lambda x.b[\#z':=x]]} \\
\\
\frac{e_1 \mapsto e'_1}{e_1 =_{\#} e_2 \mapsto e'_1 =_{\#} e_2} \quad \frac{e \mapsto e'}{\#z' =_{\#} e \mapsto \#z' =_{\#} e'} \quad \frac{}{\#z =_{\#} z \mapsto \text{True}()} \quad \frac{\#z \neq \#z'}{\#z =_{\#} z' \mapsto \text{False}()} \\
\\
\frac{e \mapsto e'}{\text{isOVar } e \mapsto \text{isOVar } e'} \quad \frac{}{\text{isOVar } \#z \mapsto \text{True}()} \quad \frac{v \neq \#z}{\text{isOVar } v \mapsto \text{False}()}
\end{array}$$

3 Basic Properties

3.1 Substitution

Lemma 12 (Basic Properties of Substitution) $\forall e, e_1, e_2 \in \mathbb{E}$.

$$1. x \neq y \wedge x \notin FV(e_2) \implies$$

$$e[x := e_1][y := e_2] = e[y := e_2][x := e_1[x := e_2]]$$

$$2. \#x \neq \#y \wedge \#x \notin FV(e_2) \implies$$

$$e[\#x := e_1][\#y := e_2] = e[\#y := e_2][\#x := e_1[\#x := e_2]]$$

Proof (Lemma 12). The proof easily follows by structural induction on the expression e . \square

3.2 Parallel Reduction

Lemma 13 (Compatibility of Parallel Reduction) $\forall C \in \mathbb{C}. \forall e_1, e_2 \in \mathbb{E}.$

$$e_1 \gg e_2 \implies C[e_1] \gg C[e_2]$$

Proof (Lemma 13). Prof is by structural induction on the context C .

1. $\llbracket [e_1] \gg [e_2] \rrbracket$
2. $\llbracket \lambda x. C \rrbracket \uparrow \frac{C[e_1] \gg C[e_2]}{(\lambda x. C)[e_1] \gg (\lambda x. C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
3. $\llbracket (C \ e) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2] \quad e \gg e}{(C \ e)[e_1] \gg (C \ e)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
4. $\llbracket (e \ C) \rrbracket \uparrow \frac{e \gg e \quad C[e_1] \gg C[e_2]}{(e \ C)[e_1] \gg (e \ C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
5. $\llbracket (e, C) \rrbracket \uparrow \frac{e \gg e \quad C[e_1] \gg C[e_2]}{(e, C)[e_1] \gg (e, C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
6. $\llbracket (C, e) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2] \quad e \gg e}{(C, e)[e_1] \gg (C, e)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
7. $\llbracket (\pi_1 C) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2]}{(\pi_1 C)[e_1] \gg (\pi_1 C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
8. $\llbracket (\pi_2 C) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2]}{(\pi_2 C)[e_1] \gg (\pi_2 C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
9. $\llbracket \cdot \rrbracket \uparrow \frac{C[e_1] \gg C[e_2]}{(\lambda^{f \in F - \{f'\}}((f_i \ x_i).e_i) ++ (f' \ x).C)[e_1] \gg (\lambda^{f \in F - \{f'\}}((f_i \ x_i).e_i) ++ (f' \ x).C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
10. $\llbracket (f \ C) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2]}{(f \ C)[e_1] \gg (f \ C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
11. $\llbracket (\#z \Rightarrow C) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2]}{(\#z \Rightarrow C)[e_1] \gg (\#z \Rightarrow C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
12. $\llbracket (\lambda(\#z \Rightarrow x).C) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2]}{(\lambda(\#z \Rightarrow x).C)[e_1] \gg (\lambda(\#z \Rightarrow x).C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
13. $\llbracket (C =_{\#} e) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2] \quad e \gg e}{(C =_{\#} e)[e_1] \gg (C =_{\#} e)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
14. $\llbracket (e = \#C) \rrbracket \uparrow \frac{e \gg e \quad C[e_1] \gg C[e_2]}{(e = \#C)[e_1] \gg (e = \#C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$
15. $\llbracket (\text{isOVar } C) \rrbracket \uparrow \frac{C[e_1] \gg C[e_2]}{(\text{isOVar } C)[e_1] \gg (\text{isOVar } C)[e_2]} \text{IH}, \gg, \llbracket \cdot \rrbracket \downarrow$

\square [e.p.]

Lemma 14 (Parallel Reduction Properties) $\forall e_1 \in \mathbb{E}.$

1. $\forall b \in \mathbb{B}, e \in \mathbb{E}. b \gg e \implies e \equiv b$
2. $\forall e \in \mathbb{E}. e \gg e.$

3. $\forall \rho. \forall e_1, e_2 \in \mathbb{E}. e_1 \rightarrow_\rho e_2 \implies e_1 \gg e_2$
4. $\forall e_2 \in \mathbb{E}. e_1 \rightarrow e_2 \implies e_1 \gg e_2$
5. $\forall e_2 \in \mathbb{E}. e_1 \gg e_2 \implies e_1 \rightarrow^* e_2$
6. $\forall e_3 \in \mathbb{E}. e_2, e_4 \in \mathbb{E}. e_1 \gg e_3, e_2 \gg e_4 \implies e_1[y := e_2] \gg e_3[y := e_4]$.

Remark 15 Note that from 1 it follows that

1. $b \rightarrow e \implies e \equiv b$
2. $!b \equiv b$
3. $b \rightarrow^* e \implies e \equiv b$

Proof (Lemma 14).

Part 1 is by a simple induction over the derivation of $b \in \mathbb{B}$. Part 2 is by a simple structural induction on e . Part 3 is by a case analysis over ρ .

1. β_1

$$(\lambda x. e) v \rightarrow_{\beta_1} e[x := v] \text{ and } \frac{e \gg e \quad v \gg v}{(\lambda x. e) v \gg e[x := v]} \gg \text{by 14.2} \Downarrow$$

2. π_1

$$\pi_1(v_1, v_2) \rightarrow_{\pi_1} v_1 \text{ and } \frac{v_1 \gg v_1}{\pi_1(v_1, v_2) \gg v_1} \gg \Downarrow$$

3. π_2

$$\pi_2(v_1, v_2) \rightarrow_{\pi_2} v_2 \text{ and } \frac{v_2 \gg v_2}{\pi_2(v_1, v_2) \gg v_2} \gg \Downarrow$$

4. β_2

$$(\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) \rightarrow_{\beta_2} e[x := \lambda x. b[\#z := x]] \text{ and } \frac{e \gg e}{(\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) \gg e[x := \lambda x. b[\#z := x]]} \gg \text{by 14.2} \Downarrow$$

5. β_3

$$(\lambda^{i \in \{k\} \cup L} (f_i x_i). e_i) (f_k v) \rightarrow_{\beta_3} e_k[x_k := v] \text{ and } \frac{e_n \gg e_n \quad v \gg v}{(\lambda^{i \in \{k\} \cup L} (f_i x_i). e_i) (f_k v) \gg e_n[x_k := v]} \gg \text{by 14.2} \Downarrow$$

6. $\#$

$$\begin{aligned} \#z =_{\#} \#z &\rightarrow_{\#} \text{True}() \text{ and } \#z =_{\#} \#z \gg \text{True} \\ \frac{\#z \neq \#z'}{\#z =_{\#} \#z' \rightarrow_{\#} \text{False}()} &\text{ and } \frac{\#z \neq \#z'}{\#z =_{\#} \#z' \gg \text{False}()} \gg \Downarrow \end{aligned}$$

7. δ_{isOVar}

$$\text{isOVar } \#z \rightarrow \text{True}() \text{ and } \frac{}{\text{isOVar } \#z \gg \text{True}}$$

8. δ_{isOVar}

$$\text{isOVar } v \rightarrow \text{False}() \text{ where } v \neq \#z, \text{ and } \frac{v \neq \#z}{\text{isOVar } v \gg \text{False}()}.$$

Proof (Property 4 of Lemma 14). If we look at the definition of \rightarrow , we notice that if $e_1 \rightarrow e_2$, then there must exist some context C , so that $e_1 = C[e']$, $e_2 = C[e'']$, and $e' \rightarrow e''$.

Thus, to show that $e_1 \rightarrow e_2 \implies e_1 \gg e_2$, it is enough to prove that for any context $C \in \mathbb{C}$, $C[e'] \rightarrow C[e''] \implies C[e'] \gg C[e'']$. This, however, follows directly from Lemma 13 (Compatibility of Parallel Reduction).

Proof (Property 5 of Lemma 14). $\forall e_1 \in \mathbb{E}. \forall e_2 \in \mathbb{E}. e_1 \gg e_2 \implies e_1 \longrightarrow^* e_2$ By induction on the height of the derivation $e_1 \gg e_2$.

1. $() \gg ()$ and $() \longrightarrow^0 ()$
2. $x \gg x$ and $x \longrightarrow^0 x$.
3. $\gg \uparrow \frac{e \gg e'}{\lambda x. e \gg \lambda x. e'} \xRightarrow{IH} \frac{e \longrightarrow^* e'}{\lambda x. e \longrightarrow^* \lambda x. e'} \text{ comp. } \longrightarrow^* \Downarrow$
4. $\gg \uparrow \frac{\frac{e_1 \gg e'_1}{e_2 \gg e'_2} \xRightarrow{IH} \frac{e_1 \longrightarrow^* e'_1}{e_2 \longrightarrow^* e'_2}}{e_1 e_2 \gg e'_1 e'_2} \xRightarrow{IH} \frac{e_1 \longrightarrow^* e'_1}{e_1 e_2 \longrightarrow^* e'_1 e'_2} \text{ comp. } \longrightarrow^* \Downarrow$
5. $\gg \uparrow \frac{\frac{v \gg v'}{e \gg e'} \xRightarrow{IH} \frac{v \longrightarrow^* v'}{e \longrightarrow^* e'}}{(\lambda x. e) v \gg e'[x := v']} \xRightarrow{IH} \frac{v \longrightarrow^* v'}{(\lambda x. e) v \longrightarrow^* (\lambda x. e') v \longrightarrow^* (\lambda x. e') v' \longrightarrow_{\beta_1} e'[x := v']} \text{ comp. } \longrightarrow^* \Downarrow$
6.
$$\gg \uparrow \frac{\frac{b \gg b'}{e \gg e'} \xRightarrow{IH} \frac{b \longrightarrow^* b'}{e \longrightarrow^* e'}}{(\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) \gg e'[x := \lambda x'. b'[\#z := x']]} \xRightarrow{IH \Downarrow IH \Downarrow} \frac{b \longrightarrow^* b'}{e \longrightarrow^* e'} \text{ comp. } \longrightarrow^* \Downarrow$$
$$\frac{(\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) \longrightarrow^* (\lambda(\#z \Rightarrow x). e') (\#z \Rightarrow b') \longrightarrow_{\beta_2} e'[x := \lambda x. b'[\#z := x]]}{(\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) \longrightarrow^* (\lambda(\#z \Rightarrow x). e') (\#z \Rightarrow b') \longrightarrow_{\beta_2} e'[x := \lambda x. b'[\#z := x]]} \text{ comp. } \longrightarrow^* \Downarrow$$
7.
$$\gg \uparrow \frac{\frac{e_k \gg e'_k}{v \gg v'} \xRightarrow{IH} \frac{e_k \longrightarrow^* e'_k}{v \longrightarrow^* v'}}{(\lambda^{f \in F \cup \{k\}} f x_f. e_f) (k v) \gg e'_k[x := v']} \xRightarrow{IH \Downarrow IH \Downarrow} \frac{e_k \longrightarrow^* e'_k}{v \longrightarrow^* v'} \text{ comp. } \longrightarrow^* \Downarrow$$
$$\frac{(\lambda^{f \in F} f x_f. e_f | k x_k. e_k) (k v) \longrightarrow^* (\lambda^{f \in F} f x_f. e_f | k x_k. e'_k) (k v') \longrightarrow_{\beta_3} e'_k[x := v']}{(\lambda^{f \in F} f x_f. e_f | k x_k. e_k) (k v) \longrightarrow^* (\lambda^{f \in F} f x_f. e_f | k x_k. e'_k) (k v') \longrightarrow_{\beta_3} e'_k[x := v']} \text{ comp. } \longrightarrow^* \Downarrow$$
8. $\gg \uparrow \frac{\frac{e_1 \gg e'_1}{e_2 \gg e'_2} \xRightarrow{IH} \frac{e_1 \longrightarrow^* e'_1}{e_2 \longrightarrow^* e'_2}}{(e_1, e_2) \gg (e'_1, e'_2)} \xRightarrow{IH} \frac{e_1 \longrightarrow^* e'_1}{(e_1, e_2) \longrightarrow^* (e'_1, e'_2)} \text{ comp. } \longrightarrow^* \Downarrow$
9. $\gg \uparrow \frac{e \gg e'}{\pi_1 e \gg \pi_1 e'} \xRightarrow{IH} \frac{e \longrightarrow^* e'}{\pi_1 e \longrightarrow^* \pi_1 e'} \text{ comp. } \longrightarrow^* \Downarrow$
10. $\gg \uparrow \frac{e \gg e'}{\pi_2 e \gg \pi_2 e'} \xRightarrow{IH} \frac{e \longrightarrow^* e'}{\pi_2 e \longrightarrow^* \pi_2 e'} \text{ comp. } \longrightarrow^* \Downarrow$
11. $\uparrow \gg \frac{v_1 \gg v'_1}{\pi_1 (v_1, v_2) \gg v'_1} \xRightarrow{IH} \frac{v_1 \longrightarrow^* v'_1}{\pi_1 (v_1, v_2) \longrightarrow_{\beta_{\pi_1}} v_1 \longrightarrow^* v'_1} \text{ comp. } \longrightarrow^* \Downarrow$
12. $\uparrow \gg \frac{v_2 \gg v'_2}{\pi_2 (v_1, v_2) \gg v'_2} \xRightarrow{IH} \frac{v_2 \longrightarrow^* v'_2}{\pi_2 (v_1, v_2) \longrightarrow_{\beta_{\pi_2}} v_2 \longrightarrow^* v'_2} \text{ comp. } \longrightarrow^* \Downarrow$
13. $\gg \uparrow \frac{e \gg e'}{f e \gg f e'} \xRightarrow{IH} \frac{e \longrightarrow^* e'}{f e \longrightarrow^* f e'} \text{ comp. } \longrightarrow^* \Downarrow$
14. $\gg \uparrow \frac{e_f \gg e'_f}{\lambda^{f \in F} (f x_f). e_f \gg \lambda^{f \in F} (f x_f). e'_f} \xRightarrow{IH} \frac{e_f \longrightarrow^* e'_f}{\lambda^{f \in F} (f x_f). e_f \longrightarrow^* \lambda^{f \in F} (f x_f). e'_f} \text{ comp. } \longrightarrow^* \Downarrow$
15. $\#z \gg \#z$ and, $\#z \longrightarrow^0 \#z$

16. $\gg \uparrow \frac{e \gg e'}{\#z \Rightarrow e \gg \#z \Rightarrow e'} \xRightarrow{IH} \frac{e \rightarrow^* e'}{\#z \Rightarrow e \rightarrow^* \#z \Rightarrow e'}$
17. $\gg \uparrow \frac{e \gg e'}{\lambda(\#z \Rightarrow x).e \gg \lambda(\#z \Rightarrow x).e'} \xRightarrow{IH} \frac{e \rightarrow^* e'}{\lambda(\#z \Rightarrow x).e \rightarrow^* \lambda(\#z \Rightarrow x).e'} \text{ comp. } \rightarrow^* \Downarrow$
18. $\gg \uparrow \frac{\begin{array}{c} e_1 \gg e'_1 \\ e_2 \gg e'_2 \end{array}}{e_1 =_{\#} e_2 \gg e'_1 =_{\#} e'_2} \xRightarrow{IH} \frac{\begin{array}{c} e_1 \rightarrow^* e'_1 \\ e_2 \rightarrow^* e'_2 \end{array}}{e_1 =_{\#} e_2 \rightarrow^* e'_1 =_{\#} e'_2} \text{ comp. } \rightarrow^* \Downarrow$
19. $\gg \uparrow \frac{\#z = \#z'}{\#z = \#z' \gg \text{False}()} \text{ and } \#z = \#z' \rightarrow_{\#} \text{False}()$
20. $\frac{\#z = \#z'}{\#z = \#z' \gg \text{True}()} \text{ and } \#z = \#z' \rightarrow_{\#} \text{True}()$
21. $\gg \uparrow \frac{e \gg e'}{\text{isOVar } e \gg \text{isOVar } e'} \xRightarrow{IH} \frac{e \rightarrow^* e'}{\text{isOVar } e \rightarrow^* \text{isOVar } e'} \text{ comp. } \rightarrow^* \Downarrow$
22. $\frac{}{\text{isOVar } \#z \gg \text{True}()} \text{ and } \text{isOVar } \#z \rightarrow_{\delta_{\text{isOVar}}} \text{True}().$
23. $\frac{v \neq \#z}{\text{isOVar } v \gg \text{False}()} \text{ and } \text{isOVar } v \rightarrow_{\delta_{\text{isOVar}}} \text{False}()$

□

Proof (Property 6 of Lemma 14). The property is stated as follows:

$$\forall e_1, e_2, e_3, e_4 \in \mathbb{E}. e_1 \gg e_3, e_2 \gg e_4 \implies e_1[y := e_2] \gg e_3[y := e_4]$$

The substitution property for parallel reduction without complexity follows directly from the substitution property for parallel reduction with complexity (Lemma 18 on page 14).

Remark 16 From Lemma 14, parts 4 and 5 above we can see that that $\gg^* = \rightarrow^*$.

Proof (Remark 16). By induction on the derivations of \rightarrow^* and \gg^* , and has two parts.

$$- e_1 \rightarrow^* e_2 \implies e_1 \gg^* e_2$$

$$\gg^* \uparrow \frac{\begin{array}{c} e_1 \rightarrow u \xRightarrow{14.4} e_1 \gg u \\ u \rightarrow^* e_2 \xRightarrow{IH} u \gg^* e_2 \end{array}}{e_1 \rightarrow^* e_2} \frac{u \gg^* e_2}{e_1 \gg^* e_2} \gg^* \Downarrow$$

$$- e_1 \gg^* e_2 \implies e_1 \rightarrow^* e_2$$

Assuming that $e_1 \gg^* e_2$, it must be the case, by definition of \gg^* , that there exists some u_1 , such that $e_1 \gg u_1$ and $u_1 \gg^* e_2$. By previous property 14.5, then $e_1 \rightarrow^* u_1$. But for that to be true, there must exist some u' such that $e_1 \rightarrow u'$ and $u' \rightarrow^* u_1$.

To show that $e_1 \rightarrow^* e_2$, there must exist some u , such that $e_1 \rightarrow u$ and $u \rightarrow^* e_2$. Let u' be this u . Now, we know that $e_1 \rightarrow u$ and $u \rightarrow^* u_1$. Since $u_1 \gg^* e_2$, then $u_1 \rightarrow^* e_2$ by the induction hypothesis. Since $u \rightarrow^* u_1$ and $u_1 \rightarrow^* e_2$, by transitivity of \rightarrow^* we have that $u \rightarrow^* e_2$. Therefore $e_1 \rightarrow^* e_2$ and we are done. □

Remark 17 (Substitution with Complexity) We have already shown that parallel reduction without complexity is equivalent (in many steps) to normal reduction (in many steps). The same result applies to parallel reduction with complexity.

3.3 Substitution Lemma for Parallel Reduction with Complexity

Lemma 18 (Substitution for Parallel Reduction with Complexity) $\forall e_4, e_5, e_6, e_7 \in \mathbb{E}, X, Y \in \mathbb{N}$.

$$e_4 \gg^X e_5 \wedge e_6 \gg^Y e_7 \implies (\exists Z \in \mathbb{N}. e_4[x := e_6] \gg^Z e_5[x := e_7] \wedge Z \leq X + \#(x, e_5)Y).$$

Proof (Lemma 18). Proof is by induction on the derivation $e_4 \gg^Y e_5$. Assumption: $e_6 \gg^Y e_7$.

1. $\frac{}{() \gg^0 ()}$ and $\exists Z = 0. ()[y := e_6] \gg^0 ()[y := e_7] \wedge Z \leq 0$.
2. $\frac{}{x \gg^0 x}$
 - (a) If $x = y$ then $\exists Z = Y. x[y := e_6] \gg^Z x[y := e_7] \wedge Z \leq Y$
 - (b) If $x \neq y$ then $\exists Z = 0. x[y := e_6] \gg^Z x[y := e_7] \wedge Z \leq 0$
3. $\frac{e_1 \gg^N e_2}{\lambda x. e_1 \gg^N \lambda x. e_2}$ By the induction hypothesis, $\exists Z_1. e_1[y := e_6] \gg^{Z_1} e_2[y := e_7] \wedge Z \leq N + \#(y, e_2)Y$ By Barendregt's assumption $x \notin FV(e_6, e_7)$, so $\exists Z = Z_1. \lambda x. e_1[x := x][y := e_6] \gg^Z \lambda x. e_2[x := x][y := e_7] \wedge Z \leq N + \#(y, \lambda x. e_2)Y$
4. $\frac{e_1 \gg^M e_3 \quad e_2 \gg^N e_4}{e_1 e_2 \gg^{M+N} e_3 e_4}$. By the induction hypothesis, we obtain the following
 - (a) $\exists Z_1. e_1[y := e_6] \gg^{Z_1} e_3[y := e_7] \wedge Z_1 \leq M + \#(y, e_3)Y$
 - (b) $\exists Z_2. e_2[y := e_6] \gg^{Z_2} e_4[y := e_7] \wedge Z_2 \leq N + \#(y, e_4)Y$
 Then, $\exists Z = Z_1 + Z_2. (e_1 e_2)[y := e_6] \gg^Z (e_3 e_4)[y := e_7] \wedge Z \leq (M + N) + \#(y, e_3 e_4)Y$
5. $\frac{e_1 \gg^M e_3 \quad v_1 \gg^N v_2}{(\lambda x. e_1) v_1 \gg^{M + \#(x, e_3)N + 1} e_3[x := v_2]}$ By the induction hypothesis, we obtain:
 - (a) $\exists Z_1. e_1[y := e_6] \gg^{Z_1} e_3[y := e_7] \wedge Z_1 \leq M + \#(y, e_3)Y$
 - (b) $\exists Z_2. v_1[y := e_6] \gg^{Z_2} v_2[y := e_7] \wedge Z_2 \leq N + \#(y, v_2)Y$
 Using the Barendregt's assumption and definition of substitution, the goal can be stated as follows: $\exists Z$.

$$(\lambda x. e_1[y := e_6]) (v_1[y := e_6]) \gg^Z e_2[x := v_2][y := e_7]$$

By property of substitution (Lemma 12) that is equivalent to: $\exists Z$.

$$(\lambda x. e_1[y := e_6]) (v[y := e_6]) \gg^Z e_2[y := e_7][x := v'[y := e_7]]$$

By Barendregt's assumption x is not free in any terms other than e_1 and e_2 , and in particular in v' . From this we can simplify the goal further, and by a series of arithmetical manipulations obtain: $\exists Z = Z_1 + \#(x, e')Z_2 +$

$$1. ((\lambda x. e_1) v)[y := e_6] \gg^Z e_2[x := v_2][y := e_7] \wedge Z \leq M + \#(x, e_2)N + 1 + \#(y, e'[x := v'])Y.$$

$$6. \frac{e_1 \gg^M e_2}{(\lambda(\#z \Rightarrow x). e_1) (\#z \Rightarrow b) \gg^{M+1} e_2[x := \lambda x. b[\#z := x]]}$$

By the induction hypothesis: $\exists Z_1. e_1[y := e_6] \gg^{Z_1} e_2[y := e_7] \wedge Z_1 \leq M + \#(y, e_2)Y$. The goal is: $\exists Z$.

$$(((\lambda \#z \Rightarrow x). e_1) (\#z \Rightarrow b)) \gg^Z e_2[x := \lambda x'. b[\#z := x']][y := e_7] \wedge Z \leq M + 1 + \#(y, e_2[x := \lambda x'. b[\#z := x']])Y$$

Since $FV(b) = \emptyset$, this can be simplified to

$$(((\lambda \#z \Rightarrow x). e_1) (\#z \Rightarrow b)) \gg^Z e_2[x := \lambda x'. b[\#z := x']][y := e_7] \wedge Z \leq M + 1 + \#(y, e_2)$$

Then further, by "permutation" of substitution:

$$(((\lambda \#z \Rightarrow x). e_1) (\#z \Rightarrow b)) \gg^Z e_2[y := e_7][x := \lambda x'. b[\#z := x']] \wedge Z \leq M + 1 + \#(y, e_2)$$

Now the goal follows easily when $Z = Z_1 + 1$.

$$7. \frac{e_k \gg^M e'_k \quad v_1 \gg^N v_2}{(\lambda^{f \in F \cup \{k\}} f \ x_f.e_f) (f_k \ v_1) \gg^{M + \#(x_k, e'_k)N + 1} e'_k[x_k := v_2]} \quad \text{By the induction hypothesis we have}$$

$$(a) \exists Z_1. e_k[y := e_6] \gg^{Z_1} e'_k[y := e_7] \wedge Z_1 \leq M + \#(y, e'_k)Y$$

$$(b) \exists Z_2. v_1[y := e_6] \gg^{Z_2} v_2[y := e_7] \wedge Z_2 \leq N + \#(y, v_2)Y$$

Using Barendregt's assumption and the definition of substitution, the goal can be restated as follows: $\exists Z$.

$$((\lambda^{i \in F \cup \{k\}} f \ x_f.e_f) (f_k \ v_1))[y := e_6] \gg^Z e'_k[x := v_2][y := e_7] \wedge Z \leq (M + \#(x, e'_k)N + 1) + \#(y, e'_k[x := v_2])Y$$

By property of substitution (Lemma 12), we obtain $\exists Z$.

$$((\lambda^{i \in F \cup \{k\}} f \ x_f.e_f) (f_k \ v_1))[y := e_6] \gg^Z e'_k[y := e_7][x := v_2[y := e_7]] \wedge Z \leq (M + \#(x, e'_k)N + 1) + \#(y, e'_k[x := v_2])Y$$

By the induction hypothesis, properties of substitution and arithmetic the above goal follows when $Z = Z_1 + \#(x, e'_k)Z_2 + 1$.

$$8. \frac{e_1 \gg^M e_3 \quad e_2 \gg^N e_4}{(e_1, e_2) \gg^{M+N} (e_3, e_4)} \quad \text{By the induction hypothesis, we obtain}$$

$$(a) \exists Z_1. e_1[y := e_6] \gg^{Z_1} e_3[y := e_7] \wedge Z_1 \leq M + \#(y, e_3)Y$$

$$(b) \exists Z_2. e_2[y := e_6] \gg^{Z_2} e_4[y := e_7] \wedge Z_2 \leq N + \#(y, e_4)Y$$

Then, it follows $\exists Z = Z_1 + Z_2. (e_1, e_2)[y := e_7] \gg^Z (e_3, e_4)[y := e_7] \wedge Z \leq (M + N) + \#(y, (e_3, e_4))Y$

$$9. \frac{e_1 \gg^M e_2}{\pi_1 \ e_1 \gg^M \pi_1 \ e_2} \quad \text{By the induction hypothesis we obtain}$$

$\exists Z_1. e_1[y := e_6] \gg^{Z_1} e_2[y := e_7] \wedge Z_1 \leq M + \#(y, e_2)Y$. Then, $\exists Z = Z_1. (\pi_1 \ e_1)[y := e_6] \gg^Z (\pi_1 \ e_2)[y := e_7] \wedge Z \leq M + \#(y, e_2)Y$.

$$10. \text{ For the derivation } \frac{e_1 \gg^M e_2}{\pi_2 \ e_1 \gg^M \pi_2 \ e_2} \quad \text{By the induction hypothesis we obtain}$$

$\exists Z_1. e_1[y := e_6] \gg^{Z_1} e_2[y := e_7] \wedge Z_1 \leq M + \#(y, e_2)Y$. Then, $\exists Z = Z_1. (\pi_2 \ e_1)[y := e_6] \gg^Z (\pi_2 \ e_2)[y := e_7] \wedge Z \leq M + \#(y, e_2)Y$.

$$11. \frac{v_1 \gg^N v'_1}{\pi_1 (v_1, v_2) \gg^{N+1} v'_1}$$

By the induction hypothesis we obtain: $\exists Z_1. v_1[y := e_6] \gg^{Z_1} v'_1[y := e_7] \wedge Z_1 \leq N + \#(y, v'_1)Y$.

Then $\exists Z = Z_1 + 1. (\pi_1 (v_1, v_2))[y := e_6] \gg^Z v'_1[y := e_7] \wedge Z \leq N + 1 + \#(y, v'_1)Y$.

$$12. \frac{v_2 \gg^N v'_2}{\pi_2 (v_1, v_2) \gg^{N+1} v'_2}$$

By the induction hypothesis we obtain: $\exists Z_1. v_2[y := e_6] \gg^{Z_1} v'_2[y := e_7] \wedge Z_1 \leq N + \#(y, v'_2)Y$.

Then $\exists Z = Z_1 + 1. (\pi_2 (v_1, v_2))[y := e_6] \gg^Z v'_2[y := e_7] \wedge Z \leq N + 1 + \#(y, v'_2)Y$.

$$13. \frac{e_1 \gg^M e_2}{f \ e_1 \gg^M f \ e_2} :$$

By the induction hypothesis $\exists Z_1. e_1[y := e_6] \gg^{Z_1} e_2[y := e_7] \wedge Z_1 \leq M + \#(y, e_2)Y$.

Then $\exists Z = Z_1. (f \ e_1)[y := e_6] \gg^Z (f \ e_2)[y := e_7] \wedge Z \leq M + \#(y, f \ e_2)Y$.

$$14. \frac{e_f \gg^{N_f} e'_f}{\lambda^{f \in F} f \ x_f.e_f \gg^N \lambda^{f \in F} f \ x_f.e'_f}$$

By the induction hypothesis $\forall f \in F$.

$$\exists Z_f. e_f[y: = e_6] \gg^{Z_f} e'_f[y: = e_7] \wedge Z_f \leq N_f + \#(y, e'_f)Y$$

Then $\exists Z = \sum_{f \in F} Z_f. (\lambda^{f \in F} f x_f. e_f[y: = e_6]) \gg^Z e'_f[y: = e_7] \wedge Z \leq \sum N_f + \sum \#(y, e'_f)Y$.

$$15. \frac{}{\#z \gg^0 \#z} \text{ and } \exists Z = 0. \#z[y: = e_6] \gg^0 \#z[y: = e_7] \wedge Z \leq 0 + 0 \cdot Y$$

$$16. \frac{e_1 \gg^M e_2}{(\#z \Rightarrow e_1) \gg^M (\#z \Rightarrow e_2)} \text{ By the induction hypothesis, } \exists Z_1. e_1[y: = e_6] \gg^{Z_1} e_2[y: = e_7] \wedge M + \#(y, e_2)Y. \text{ Then,}$$

$$\exists Z = Z_1. (\#z \Rightarrow e_1)[y: = e_6] \gg^Z (\#z \Rightarrow e_2)[y: = e_7] \wedge Z \leq M + \#(y, \#z \Rightarrow e_2).$$

$$17. \frac{e_1 \gg^N e_2}{\lambda(\#z \Rightarrow x). e_1 \gg^N \lambda(\#z \Rightarrow x). e_2} \text{ By the induction hypothesis, we obtain } \exists Z_1. e_1[y: = e_6] \gg^{Z_1} e_2[y: = e_7] \wedge Z_1 \leq$$

$$N + \#(y, e_2).$$

Then $\exists Z = Z_1. (\#z \Rightarrow e_1)[y: = e_6] \gg^Z (\#z \Rightarrow e_2)[y: = e_7] \wedge Z \leq N + \#(y, \#z \Rightarrow e_2)Y$

$$18. \frac{}{\#z = \# \#z \gg^1 \text{True}()} , \exists Z = 1. (\#z = \# z)[y: = e_6] \gg^Z \text{True}()[y: = e_7] \wedge Z \leq 1 + \#(y, \text{True}())$$

$$19. \frac{}{\#z \neq \#z' \gg^1 \text{False}()} , \exists Z = 1. (\#z \neq \# z')[y: = e_6] \gg^Z \text{False}()[y: = e_7] \wedge Z \leq 1 + \#(y, \text{False}())$$

$$20. \frac{e_1 \gg^M e_3 \quad e_2 \gg^N e_4}{e_1 = \# e_2 \gg^{M+N} e_3 = \# e_4}$$

By the induction hypothesis we obtain

$$(a) \exists Z_1. e_1[y: = e_6] \gg^{Z_1} e_3[y: = e_7] \wedge Z_1 \leq M + \#(y, e_3)Y$$

$$(b) \exists Z_2. e_2[y: = e_6] \gg^{Z_2} e_4[y: = e_7] \wedge Z_2 \leq N + \#(y, e_4)Y.$$

Then $\exists Z = Z_1 + Z_2. (e_1 = \# e_2)[y: = e_6] \gg^Z (e_3 = \# e_4)[y: = e_7] \wedge Z \leq (M + N) + \#(y, (e_e = \# e_4))Y$.

$$21. \frac{e \gg^X e'}{\text{isOVar } e \gg^X \text{isOVar } e'} \text{ By the induction hypothesis, } \exists Z_1. e[y: = e_6] \gg^{Z_1} e'[y: = e_6] \wedge Z_1 \leq X + \#(y, e')Y. \text{ Then,}$$

$\exists Z = Z_1. (\text{isOVar } e)[y: = e_6] \gg^Z (\text{isOVar } e')[y: = e_7] \wedge Z \leq X + \#(y, \text{isOVar } e')$.

$$22. \frac{}{\text{isOVar } \#z \gg^1 \text{True}()} \text{ and } \exists Z = 1. (\text{isOVar } \#z)[y: = e_6] \gg^Z \text{True}()[y: = e_7] \wedge Z \leq 1 + \#(y, \text{True}())Y.$$

$$23. \frac{v \neq \#z}{\text{isOVar } v \gg^1 \text{False}()} \text{ Then, } \exists Z = 1. (\text{isOVar } v)[y: = e_6] \gg^Z \text{False}()[y: = e_7] \wedge Z \leq 1 + \#(y, \text{False}())Y.$$

3.4 Big-Step Semantics

Lemma 19 (Basic Property of Big-Step Semantics) *If $e \hookrightarrow e'$ then $e' \in \mathbb{V}$.*

Proof. Proof is straightforward by induction over the height of the derivation. \square [e.p.]

3.5 Classes

Lemma 20 (Basic Properties of Classes)

1. $\mathbb{V}, \mathbb{W}, \mathbb{S} \subseteq \mathbb{E}$
2. $\mathbb{V}, \mathbb{W}, \mathbb{S}$ partition \mathbb{E} .

Proof. Proof is by structural induction on e , and then by pattern matching on e . The following table summarizes this rather tedious proof:

$e \in \mathbb{E}$	$\in \mathbb{V}$	$\in \mathbb{W}$	$\in \mathbb{S}$
$()$	Yes	No	No
x	No	No	Yes
$\lambda x.e$	Yes	No	No
$(\lambda x.e) v$	No	Yes	No
$(\lambda(\#z \Rightarrow x).e) (\#z \Rightarrow b)$	No	Yes	No
$(\lambda^{J \in F} f x_f.e_f) (f v)$	No	Yes	No
$w e$	No	Yes	No
$v w$	No	Yes	No
$(\mathbb{V} \setminus \lambda^e) v$	No	No	Yes
$s_1 e$	No	No	Yes
$v s$	No	No	Yes
$(\lambda(\#z \Rightarrow x).e) (\mathbb{V} \setminus \#z \Rightarrow b)$	No	No	Yes
$(\lambda^{J \in F} f x_f.e_f) (\mathbb{V} \setminus (f v))$	No	No	Yes
Exhaustive/Nonoverlapping: Yes			
(v_1, v_2)	Yes	No	No
(w, e)	No	Yes	No
(v, w)	No	Yes	No
(s, e)	No	No	Yes
(v, s)	No	No	Yes
Exhaustive/Nonoverlapping: Yes			
$\pi_1 (v_1, v_2)$	No	Yes	No
$\pi_1 w$	No	Yes	No
$\pi_1 (\mathbb{V} \setminus (v_1, v_2))$	No	No	Yes
$\pi_1 s$	No	No	Yes
Exhaustive/Nonoverlapping: Yes			

$e \in \mathbb{E}$	$\in \mathbb{V}$	$\in \mathbb{W}$	$\in \mathbb{S}$
$\pi_2 (v_1, v_2)$	No	Yes	No
$\pi_2 w$	No	Yes	No
$\pi_2 (\mathbb{V} \setminus (v_1, v_2))$	No	No	Yes
$\pi_2 s$	No	No	Yes
Exhaustive/Nonoverlapping: Yes			
$f v$	Yes	No	No
$f w$	No	Yes	No
$f s$	No	No	Yes
Exhaustive/Nonoverlapping: Yes			
$\lambda^{J \in F} (f x_f).e_f$	Yes	No	No
$\#z$	Yes	No	No
$\#z \Rightarrow v$	Yes	No	No
$\#z \Rightarrow w$	No	Yes	No
$\#z \Rightarrow s$	No	No	Yes
Exhaustive/Nonoverlapping: Yes			
$\lambda(\#z \Rightarrow x).e$	Yes	No	No
$\#z = \# \#z$	No	Yes	No
$w = \# e$	No	Yes	No
$\#z = \# w$	No	Yes	No
$(\mathbb{V} \setminus \#z) = \# e$	No	No	Yes
$\#z = \# (\mathbb{V} \setminus \#z)$	No	No	Yes
$s = \# e$	No	No	Yes
$\#z = \# s$	No	No	Yes
Exhaustive/Nonoverlapping: Yes			
isOVar v	No	Yes	No
isOVar w	No	Yes	No
isOVar s	No	No	Yes
Exhaustive/Nonoverlapping: Yes			

□[e.p.]

3.6 Parallel Reduction and Classes

There is a sense in which parallel reduction should respect the classes. The following lemma explicates these properties.

Lemma 21 (Parallel Reduction and Classes)

1. $\forall e \in \mathbb{E}, v \in \mathbb{V}. v \gg^M e \implies e \in V$
2. $\forall e \in \mathbb{E}, s \in \mathbb{S}. s \gg^M e \implies e \in S$
3. $\forall e \in \mathbb{E}, w \in \mathbb{W}. e \gg^M w \implies e \in W$.

Proof (Lemma 21.1). By structural induction on $v \in \mathbb{V}$.

1. $() \in \mathbb{V}$ and $() \gg () \in \mathbb{V}$
2. $\frac{e \gg e'}{\lambda x.e \gg \lambda x.e'}$ and $\lambda x.e' \in \mathbb{V}$.

3. $\gg \uparrow \frac{\frac{v_1 \gg e_1}{v_2 \gg e_2}}{(v_1, v_2) \gg (e_1, e_2)} \xrightarrow{IH} \frac{\frac{e_1 \in \mathbb{V}}{e_2 \in \mathbb{V}}}{(e_1, e_2) \in \mathbb{V}} \epsilon_v \downarrow$
4. $\gg \uparrow \frac{v \gg e}{f v \gg f e} \xrightarrow{IH} \frac{e \in \mathbb{V}}{f v' \in \mathbb{V}} \epsilon_v \downarrow$
5. $\lambda^{i \in L} f_i x_i.e_i \gg \lambda^{i \in L} f_i x_i.e'_i$ and $\lambda^{i \in L} f_i x_i.e'_i \in \mathbb{V}$
6. $\#z \in \mathbb{V}$ and $\#z \gg \#z \in \mathbb{V}$
7. $\gg \uparrow \frac{v \gg v'}{\#z \Rightarrow v \gg \#z \Rightarrow v'} \xrightarrow{IH} \frac{v' \in \mathbb{V}}{\#z \Rightarrow v' \in \mathbb{V}} \epsilon_v \downarrow$
8. $\lambda(\#z \Rightarrow x).e' \in \mathbb{V}$ and $\lambda(\#z \Rightarrow x).e \gg \lambda(\#z \Rightarrow x).e' \in \mathbb{V}$

□

Proof (Lemma 21.2).

$$\forall e \in \mathbb{E}, s \in S. s \overset{M}{\gg} e \implies e \in S$$

By structural induction on $s \in \mathbb{S}$.

1. $x \in \mathbb{S}$ and $x \gg x$ and $x \in \mathbb{S}$.
2. $\gg \uparrow \frac{\frac{e_1 \gg e_2}{s_1 \gg e_3} \xrightarrow{IH} \frac{e_2 \in \mathbb{E}}{e_3 \in \mathbb{S}}}{s_1 e_1 \gg e_3 e_2} \xrightarrow{IH} \frac{e_3 \in \mathbb{S}}{e_3 e_2 \in \mathbb{S}} \epsilon_s \downarrow$
3. $\gg \uparrow \frac{\frac{v \gg e_1}{s \gg e_2} \xrightarrow{21.1} \frac{e_1 \in \mathbb{V}}{e_2 \in \mathbb{S}}}{v s \gg e_1 e_2} \xrightarrow{IH} \frac{e_2 \in \mathbb{S}}{e_1 e_2 \in \mathbb{S}} \epsilon_s \downarrow$
4. $\gg \uparrow \frac{\frac{(\lambda(\#z \Rightarrow x).e) \gg (\lambda(\#z \Rightarrow x).e')}{v \gg v' \quad v \in \mathbb{V} \setminus (\#z \Rightarrow b)} \xrightarrow{\gg, 21.1} \frac{v' \in \mathbb{V} \setminus (\#z \Rightarrow b)}{(\lambda(\#z \Rightarrow x).e) v' \in \mathbb{S}} \epsilon_s \downarrow}{(\lambda(\#z \Rightarrow x).e) v \gg (\lambda(\#z \Rightarrow x).e') v'}$
5. $\gg \uparrow \frac{\frac{v \gg v'}{s \gg s'} \xrightarrow{IH} \frac{s' \in \mathbb{S}}{(v', s') \in \mathbb{S}} \epsilon_s \downarrow}{(v, s) \gg (v', s')}$
6. $\gg \uparrow \frac{\frac{e \gg e'}{s_1 \gg s'_1} \xrightarrow{IH} \frac{e' \in \mathbb{E}}{s'_1 \in \mathbb{S}} \epsilon_s \downarrow}{(s_1, e) \gg (s'_1, e')}$
7. $\gg \uparrow \frac{s \gg s'}{\pi_n s \gg_n s'} \xrightarrow{IH} \frac{s' \in \mathbb{S}}{\pi_n s' \in \mathbb{S}} \epsilon_s \downarrow$
8. $\gg \uparrow \frac{v \gg v' \quad v \in (\mathbb{V} \setminus (v, v))}{\pi_n v \gg \pi_n v'} \xrightarrow{\gg} \frac{v' \in (\mathbb{V} \setminus (v, v))}{\pi_n v' \in \mathbb{S}} \epsilon_s \downarrow$
9. $\gg \uparrow \frac{s \gg s'}{f s \gg f s'} \xrightarrow{IH} \frac{s' \in \mathbb{S}}{f s' \in \mathbb{S}} \epsilon_s \downarrow$
10. $\gg \uparrow \frac{\frac{\lambda^{f \in F} f x_f.e_f \gg \lambda^{f \in F} f x_f.e'_f}{v \gg v' \quad v \in (\mathbb{V} \setminus f v)} \xrightarrow{IH} \frac{v' \in (\mathbb{V} \setminus f v)}{(\lambda^{f \in F} f x_f.e'_f) v' \in \mathbb{S}} \epsilon_s \downarrow}{(\lambda^{f \in F} f x_f.e_f) v \gg (\lambda^{f \in F} f x_f.e'_f) v'}$
11. $\gg \uparrow \frac{s \gg s'}{\#z \Rightarrow s \gg \#z \Rightarrow s'} \xrightarrow{IH} \frac{s' \in \mathbb{S}}{\#z \Rightarrow s' \in \mathbb{S}} \epsilon_s \downarrow$
12. $\gg \uparrow \frac{v \gg v' \quad v \in (\mathbb{V} \setminus \mathbb{E})}{\#z \Rightarrow v \gg \#z \Rightarrow v'} \xrightarrow{IH} \frac{v' \in (\mathbb{V} \setminus \mathbb{E})}{\#z \Rightarrow v' \in \mathbb{S}} \epsilon_s \downarrow$

$$\begin{array}{l}
13. \triangleright \uparrow \frac{e \gg e' \quad v \gg v' \quad v' \in (\mathbb{V} \setminus \#z)}{v = e \gg v' = e'} \xRightarrow{\gg} \frac{v' \in (\mathbb{V} \setminus \#z)}{v' = e' \in \mathbb{S}} \epsilon_S \Downarrow \\
14. \triangleright \uparrow \frac{\#z \gg \#z \quad v \gg v' \quad v \in (\mathbb{V} \setminus \#z)}{\#z = v \gg \#z = v'} \xRightarrow{\gg, \epsilon_V} \frac{v' \in (\mathbb{V} \setminus \#z)}{\#z = v' \in \mathbb{S}} \epsilon_S \Downarrow \\
15. \triangleright \uparrow \frac{e \gg e' \quad s \gg s'}{s = e \gg s'_1 = s'_2} \xRightarrow{IH} \frac{e' \in \mathbb{E} \quad s' \in \mathbb{S}}{s' = e' \in \mathbb{S}} \epsilon_S \Downarrow \\
16. \triangleright \uparrow \frac{s \gg s'}{\#z = \#s \gg \#z = \#s'} \xRightarrow{IH} \frac{s' \in \mathbb{S}}{\#z = \#s' \in \mathbb{S}} \epsilon_S \Downarrow \\
17. \triangleright \uparrow \frac{v_2 \gg e_2 \quad v_2 \in \mathbb{V} \quad v_1 \gg e_1 \quad v_1 \in (\mathbb{V} \setminus \lambda^e)}{v_1 v_2 \gg e_1 e_2} \xRightarrow{\gg, 21.1} \frac{e_2 \in \mathbb{V} \quad e_1 \in (\mathbb{V} \setminus \lambda^e)}{e_1 e_2 \in \mathbb{S}} \epsilon_S \Downarrow \\
18. \triangleright \uparrow \frac{s \gg e}{\text{isOVar } s \gg \text{isOVar } e} \xRightarrow{IH} \frac{e \in \mathbb{S}}{\text{isOVar } e \in \mathbb{S}} \epsilon_S \Downarrow
\end{array}$$

□

Proof (Lemma 21.3).

$$\forall e \in \mathbb{E}, w \in W. e \xRightarrow{M} w \implies e \in W.$$

Property 3 follows directly from the previous two properties. □

3.7 Left Reduction

Lemma 22 (Left Reduction and Classes)

1. $\forall w \in W. (\exists e' \in \mathbb{E}. w \mapsto e')$
2. $\forall e \in \mathbb{E}. (\exists e' \in \mathbb{E}. e \mapsto e') \implies e \in W$
3. $\forall v \in \mathbb{V}. \neg(\exists e' \in \mathbb{E}. v \mapsto e')$
4. $\forall s \in S. \neg(\exists e' \in \mathbb{E}. s \mapsto e').$

Proof (Lemma 22). We only need to prove the first two, and the second two follow. The first one is by straightforward induction on the judgement $e \in \mathbb{W}$. The second is also by straightforward induction on the derivation $e \mapsto e'$.

$$\forall w \in W. (\exists e' \in \mathbb{E}. w \mapsto e')$$

By structural induction on w .

1. $(\lambda x. e) v \mapsto e[x := v]$ and $e[x := v] \in \mathbb{E}$
2. $(\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) \mapsto e[x := \lambda x'. b[\#z := x']]$ and $e[x := \lambda x'. b[\#z := x']] \in \mathbb{E}$
3. $\mapsto \uparrow \frac{w \mapsto w'}{w e \mapsto w' e} \xRightarrow{IH, \epsilon_E} \Downarrow$
4. $\mapsto \uparrow \frac{w \mapsto w'}{v w \mapsto v w'} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{v w' \in \mathbb{E}} \epsilon_E \Downarrow$
5. $\mapsto \uparrow \frac{w \mapsto w'}{f w \mapsto f w'} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{f w' \in \mathbb{E}} \epsilon_E \Downarrow$
6. $\mapsto \uparrow \frac{w \mapsto w'}{\#z \Rightarrow w \mapsto \#z \Rightarrow w'} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{\#z \Rightarrow w' \in \mathbb{E}} \epsilon_E \Downarrow$

7. $\rightarrow \uparrow \frac{w \mapsto w'}{(w, e) \mapsto (w', e)} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{(w', e) \in \mathbb{E}} \in \mathbb{E} \Downarrow$
8. $\rightarrow \uparrow \frac{w \mapsto w'}{(v, w) \mapsto (v, w')} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{(v, w') \in \mathbb{E}} \in \mathbb{E} \Downarrow$
9. $\rightarrow \uparrow \frac{w \mapsto w'}{w = e \mapsto w' = e} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{w' = e \in \mathbb{E}} \in \mathbb{E} \Downarrow$
10. $\rightarrow \uparrow \frac{w \mapsto w'}{\#z = w \mapsto \#z = w'} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{\#z = w' \in \mathbb{E}} \in \mathbb{E} \Downarrow$
11. $\#z = \#z' \mapsto \text{True}() \text{ and } \text{True} \in \mathbb{E}$
12. $\rightarrow \uparrow \frac{w \mapsto w'}{\pi_n w \mapsto \pi_n w'} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{\pi_n w' \in \mathbb{E}} \in \mathbb{E} \Downarrow$
13. $\pi_1(v_1, v_2) \mapsto v_1 \text{ and } v_1 \in \mathbb{E}$
14. $\pi_2(v_1, v_2) \mapsto v_2 \text{ and } v_2 \in \mathbb{E}$
15. $\text{isOVar } \#z \mapsto \text{True}() \text{ and } \text{True}() \in \mathbb{E}$
16. $\text{isOVar } v \mapsto \text{False}() \text{ where } v \neq \#z, \text{ and } \text{False}() \in \mathbb{E}$
17. $\rightarrow \uparrow \frac{w \mapsto w'}{\text{isOVar } w \mapsto \text{isOVar } w'} \xRightarrow{IH} \frac{w' \in \mathbb{E}}{\text{isOVar } w' \in \mathbb{E}} \in \mathbb{E} \Downarrow$

Part 2 is proven by induction on the height of derivations of $e \mapsto e'$, and by case analysis on $e \mapsto e'$.

$$\forall e \in \mathbb{E}. (\exists e' \in \mathbb{E}. e \mapsto e') \implies e \in \mathbb{W}$$

1. $\frac{}{(\lambda x. e) v \mapsto e[x := v]} \text{ and } (\lambda x. e) v \in \mathbb{W}$
2. $\rightarrow \uparrow \frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \xRightarrow{IH} \frac{e_1 \in \mathbb{W}}{e_1 e_2 \in \mathbb{W}} \in \mathbb{W} \Downarrow$
3. $\rightarrow \uparrow \frac{e \mapsto e'}{v e \mapsto v e'} \xRightarrow{IH} \frac{e \in \mathbb{W}}{v e \in \mathbb{W}}$
4. $\rightarrow \uparrow \frac{e_1 \mapsto e'_1}{(e_1, e_2) \mapsto (e'_1, e_2)} \xRightarrow{IH} \frac{e_1 \in \mathbb{W}}{(e_1, e_2) \in \mathbb{W}} \in \mathbb{W} \Downarrow$
5. $\rightarrow \uparrow \frac{e \mapsto e'}{(v, e) \mapsto (v, e')} \xRightarrow{IH} \frac{e \in \mathbb{W}}{(v, e) \in \mathbb{W}} \in \mathbb{W} \Downarrow$
6. $\rightarrow \uparrow \frac{e \mapsto e'}{\pi_1 e \mapsto \pi_1 e'} \xRightarrow{IH} \frac{e \in \mathbb{W}}{\pi_1 e \in \mathbb{W}} \in \mathbb{W} \Downarrow$
7. $\rightarrow \uparrow \frac{e \mapsto e'}{\pi_2 e \mapsto \pi_2 e'} \xRightarrow{IH} \frac{e \in \mathbb{W}}{\pi_2 e \in \mathbb{W}} \in \mathbb{W} \Downarrow$
8. $\frac{}{\pi_1(v_1, v_2) \mapsto v_1} \text{ and } \pi_1(v_1, v_2) \in \mathbb{W}$
9. $\frac{}{\pi_2(v_1, v_2) \mapsto v_2} \text{ and } \pi_2(v_1, v_2) \in \mathbb{W}$
10. $\frac{}{(\lambda^{f \in F \cup \{k\}} (f x_f). e_f) (k v) \mapsto e_k[x_k := v]} \text{ , and } (\lambda^{f \in F \cup \{k\}} (f x_f). e_f) (k v) \in \mathbb{W}$
11. $\rightarrow \uparrow \frac{e \mapsto e'}{f e \mapsto f e'} \xRightarrow{IH} \frac{e \in \mathbb{W}}{f e \in \mathbb{W}} \in \mathbb{W} \Downarrow$
12. $\rightarrow \uparrow \frac{e \mapsto e'}{\#z \Rightarrow e \mapsto \#z \Rightarrow e'} \xRightarrow{IH} \frac{e \in \mathbb{W}}{\#z \Rightarrow e \in \mathbb{W}} \in \mathbb{W} \Downarrow$
13. $\frac{}{(\lambda(\#z \Rightarrow x). e) (\#z' \Rightarrow b) \mapsto e[x := \lambda x'. b[\#z' := x']]} \text{ and } (\lambda(\#z \Rightarrow x). e) (\#z' \Rightarrow b) \in \mathbb{W}$
14. $\rightarrow \uparrow \frac{e_1 \mapsto e'_1}{e_1 =_{\#} e_2 \mapsto e'_1 =_{\#} e_2} \xRightarrow{IH} \frac{e_1 \in \mathbb{W}}{e_1 =_{\#} e_2 \in \mathbb{W}} \in \mathbb{W} \Downarrow$

15. $\mapsto \Uparrow \frac{e \mapsto e'}{\#z' =_{\#} e \mapsto \#z' =_{\#} e'} \xRightarrow{IH} \frac{e \in \mathbb{W}}{\#z' =_{\#} e \in \mathbb{W}} \epsilon_{\mathbb{W}} \Downarrow$
16. $\frac{}{\#z =_{\#} z \mapsto \text{True}()} \text{ and } \#z =_{\#} \#z \in \mathbb{W}$
17. $\frac{\#z \neq \#z'}{\#z =_{\#} z' \mapsto \text{False}()} \text{ and } \#z =_{\#} \#z \in \mathbb{W}$
18. $\frac{}{\text{isOVar } \#z \mapsto \text{True}()} \text{ and isOVar } \#z \in \mathbb{W}.$
19. $\frac{v \neq \#z}{\text{isOVar } v \mapsto \text{False}()} \text{ and isOVar } v \in \mathbb{W}.$
20. $\mapsto \Uparrow \frac{e \mapsto e'}{\text{isOVar } e \mapsto \text{isOVar } e'} \xRightarrow{IH} \frac{e \in \mathbb{W}}{\text{isOVar } e \in \mathbb{W}} \epsilon_{\mathbb{W}} \Downarrow$

□[e.p]

Remark 23 (Left Reduction Determinism) *From the above it easily follows that for any expression $e \in \mathbb{E}$, if $e \mapsto e'$, then there is only one derivation by which $e \mapsto e'$. □*

4 General Part of Confluence

The Church-Rosser theorem [1] for \longrightarrow follows from Takahashi's property [9] (Theorem 27). The statement of Takahashi's property uses the notion of a *complete development*.

4.1 Parallel Reduction is Diamond

Lemma 24 (Parallel Reduction is Diamond) $\forall e_1, e, e_2 \in \mathbb{E}.$

$$e_1 \ll e \gg e_2 \implies (\exists e' \in \mathbb{E}. e_1 \gg e' \ll e_2).$$

4.2 Takahashi's Property

Proof. Take $e' = !e$ and use Takahashi's property (Theorem 27). □

4.3 Main Confluence Result

Theorem 25 (Main Confluence Result) .

$$\forall e_1, e, e_2 \in \mathbb{E}. \quad e_1 \longleftarrow^* e \longrightarrow^* e_2 \implies \exists e' \in \mathbb{E}. \quad e_1 \longrightarrow^* e' \longleftarrow^* e_2$$

Proof (Theorem 25). Follows directly from Lemma 24. □

5 Special Part of Confluence

Remark 26 *By a simple induction on e , we can see that $e \gg !e$.*

5.1 Takahashi's Property

Theorem 27 (Takahashi's Property) $\forall e_1, e_2 \in \mathbb{E}.$

$$e_1 \gg e_2 \implies e_2 \gg !e_1.$$

Proof (Takahashi's property for \gg).

$$\forall e_1, e_2 \in \mathbb{E} \quad e_1 \gg e_2 \implies e_2 \gg! e_2$$

By induction on the height of the derivation $e_1 \gg e_2$.

1. $() \gg ()$ and $() \gg! ()$.
2. For $x \gg x$ and $x \gg! x$
3. $\gg \uparrow \frac{e \gg e' \xRightarrow{IH} e' \gg! e}{\lambda x. e \gg \lambda x. e' \quad (\lambda x. e') \gg! (\lambda x. e)} \gg \downarrow$
4. $\gg \uparrow \frac{v \gg v' \xRightarrow{IH} v' \gg! v}{e \gg e' \quad e'[x := v'] \gg! ((\lambda x. e) v)} \gg \downarrow$ 14.6. $\gg \downarrow$
5. $\gg \uparrow \frac{e \gg e'}{(\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b) \gg e'[x := \lambda x'. b[\#z := x']]} \xRightarrow{IH} \frac{e' \gg! e}{e'[x := \lambda x'. b[\#z := x']] \gg! ((\lambda(\#z \Rightarrow x). e) (\#z \Rightarrow b))} \gg \downarrow$ 1.14
6. $\gg \uparrow \frac{v \gg v' \quad e_k \gg e'_k \xRightarrow{IH} v' \gg! v \quad e'_k \gg! e_k}{(\lambda^{f \in F \cup \{k\}} f x_f. e_f) (k v) \gg e'_k[x := v']} \xRightarrow{IH} \frac{v' \gg! v \quad e'_k \gg! e_k}{e'_k[x := v'] \gg! ((\lambda^{f \in F \cup \{k\}} f x_f. e_f) (k v))} \gg \downarrow$ 1.14.6
7. $\gg \uparrow \frac{e_2 \gg e'_2 \xRightarrow{IH} e'_2 \gg! e_1 \quad e_1 \gg e'_1 \xRightarrow{IH} e'_1 \gg! e_1}{e_1 e_2 \gg e'_1 e'_2 \quad (e'_1 e'_2) \gg! (e_1 e_2)} \gg \downarrow$
8. $\gg \uparrow \frac{e_2 \gg e'_2 \quad e_1 \gg e'_1 \xRightarrow{IH} e'_2 \gg! e_2 \quad e'_1 \gg! e_1}{(e_1, e_2) \gg (e'_1, e'_2) \quad (e'_1, e'_2) \gg! (e_1, e_2)} \gg \downarrow$
9. $\gg \uparrow \frac{e \gg e' \xRightarrow{IH} e' \gg! e}{\pi_1 e \gg \pi_1 e' \quad (\pi_1 e') \gg! (\pi_1 e)} \gg \downarrow$
10. $\gg \uparrow \frac{e \gg e' \xRightarrow{IH} e' \gg! e}{\pi_2 e \gg \pi_2 e' \quad (\pi_2 e') \gg! (\pi_2 e)} \gg \downarrow$
11. $\gg \uparrow \frac{v_1 \gg v'_1 \xRightarrow{IH} v'_1 \gg! v_1}{\pi_1 (v_1, v_2) \gg v'_1 \quad v'_1 \gg! (\pi_1 (v_1, v_2))} \gg \downarrow$
12. $\gg \uparrow \frac{v_2 \gg v'_2 \xRightarrow{IH} v'_2 \gg! v_2}{\pi_2 (v_1, v_2) \gg v'_2 \quad v'_2 \gg! (\pi_2 (v_1, v_2))} \gg \downarrow$
13. $\gg \uparrow \frac{e \gg e' \xRightarrow{IH} e' \gg! e}{f e \gg f e' \quad f e' \gg! (f e)} \gg \downarrow$
14. $\gg \uparrow \frac{e_f \gg e'_f \xRightarrow{IH} e'_f \gg! e_f}{\lambda^{f \in F} f x_f. e_f \gg \lambda^{f \in F} f x_f. e'_f \quad \lambda^{f \in F} f x_f. e'_f \gg! (\lambda^{f \in F} f x_f. e_f)} \gg \downarrow$
15. $\#z \gg \#z$ and $\#z \gg! \#z$
16. $\gg \uparrow \frac{e \gg e' \xRightarrow{IH} e' \gg! e}{\#z \Rightarrow e \gg \#z \Rightarrow e' \quad \#z \Rightarrow e' \gg! (\#z \Rightarrow e)} \gg \downarrow$
17. $\gg \uparrow \frac{e \gg e' \xRightarrow{IH} e' \gg! e}{\lambda(\#z \Rightarrow x). e \gg \lambda(\#z \Rightarrow x). e' \quad \lambda(\#z \Rightarrow x). e' \gg! (\lambda(\#z \Rightarrow x). e)} \gg \downarrow$
18. $\gg \uparrow \frac{e_1 \gg e'_1 \quad e_2 \gg e'_2 \xRightarrow{IH} e'_1 \gg! e_1 \quad e'_2 \gg! e_2}{e_1 = e_2 \gg e'_1 = e'_2 \quad (e'_1 = e'_2) \gg! (e_1 = e_2)} \gg \downarrow$
19. $\#z = \#z \gg \text{True}()$ and $\text{True}() \gg! (\#z = \#z)$
20. $\#z \neq \#z' \quad \#z = \#z \gg \#z \neq \#z' \gg \text{False}()$ and $\text{False}() \gg! (\#z = \#z')$

21. $\text{isOVar } \#z \gg \text{True}$ and $\text{True}() \gg !(\text{isOVar } \#z)$.
22. $\frac{v \neq \#z}{\text{isOVar } v \gg \text{False}()}$, and $\text{False}() \gg !(\text{isOVar } v)$.
23. $\gg \uparrow \frac{e \gg e'}{\text{isOVar } e \gg \text{isOVar } e'} \xRightarrow{IH} \frac{e' \gg !e}{\text{isOVar } e' \gg !(\text{isOVar } e)} \gg \downarrow$

□[e.p.]

6 General Part of Computational Adequacy of Reduction Semantics

6.1 Main Soundness Theorem

Definition 28 (Observational Equivalence)

$$e_1 \approx e_2 \equiv \forall C \in \mathbb{C}. C[e_1] \Downarrow \Leftrightarrow C[e_2] \Downarrow$$

Definition 29 (Termination)

$$e_1 \Downarrow \equiv \exists v \in \mathbb{V}. e_1 \hookrightarrow v$$

Theorem 30 (Soundness Theorem)

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies e_1 \approx e_2$$

Proof (Theorem 30). By the definition of \approx , to prove our goal

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies e_1 \approx e_2$$

is to prove

$$C \in \mathbb{C}. \forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \wedge C[e_1], C[e_2] \in \mathbb{E} \implies (C[e_1] \Downarrow \Leftrightarrow C[e_2] \Downarrow).$$

Noting that by the compatibility of \longrightarrow , we know that $C \in \mathbb{C}. \forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies C[e_1] \longrightarrow C[e_2]$, it is sufficient to prove a stronger statement:

$$C \in \mathbb{C}. \forall e_1, e_2 \in \mathbb{E}. C[e_1] \longrightarrow C[e_2] \wedge C[e_1], C[e_2] \in \mathbb{E} \implies (C[e_1] \Downarrow \Leftrightarrow C[e_2] \Downarrow).$$

Noting further that $C \in \mathbb{C}. \forall a, b \in \mathbb{E}. a \equiv C[b] \in \mathbb{E} \implies a \in \mathbb{E}$, it is sufficient to prove an even stronger statement:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies (e_1 \Downarrow \Leftrightarrow e_2 \Downarrow).$$

This goal can be broken down into two parts:

S1

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies (e_1 \Downarrow \implies e_2 \Downarrow),$$

and

S2

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies (e_2 \Downarrow \implies e_1 \Downarrow).$$

Let us consider S1. By definition of termination, it says:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies ((\exists v \in \mathbb{V}. e_1 \hookrightarrow v) \implies (\exists v \in \mathbb{V}. e_2 \hookrightarrow v)).$$

We will show that big-step evaluation is included in reduction (Lemma 34). Thus, to prove S2 it is enough to prove:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies ((\exists v \in \mathbb{V}. e_1 \longrightarrow^* v) \implies (\exists v \in \mathbb{V}. e_2 \hookrightarrow v)).$$

Confluence (Theorem 25) tell us that any two reduction paths are joinable, so we can weaken our goal as follows:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies ((\exists v \in \mathbb{V}. e_3 \in \mathbb{E}. e_1 \longrightarrow^* v \longrightarrow^* e_3 \wedge e_2 \longrightarrow^* e_3) \implies (\exists v \in \mathbb{V}. e_2 \hookrightarrow v))$$

We will show (Lemmas 21 and Remark 16) that any reduction that starts from a value can only lead to a value (at the same level). Thus we can weaken further:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies ((\exists v, v_3 \in \mathbb{V}. e_1 \longrightarrow^* v \longrightarrow^* v_3 \wedge e_2 \longrightarrow^* v_3) \implies (\exists v \in \mathbb{V}. e_2 \hookrightarrow v))$$

In other words, we already know that e_2 reduces to a value, and the question is really whether it *evaluates* to a value. Formally:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies ((\exists v_3 \in \mathbb{V}. e_2 \longrightarrow^* v_3) \implies (\exists v \in \mathbb{V}. e_2 \hookrightarrow v)).$$

In fact, the original assumption is no longer necessary, and we will prove:

T1

$$\forall e_2 \in \mathbb{E}. ((\exists v_3 \in \mathbb{V}. e_2 \longrightarrow^* v_3) \implies (\exists v \in \mathbb{V}. e_2 \hookrightarrow v)).$$

Now consider S2. By definition of termination, it says:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies ((\exists v \in \mathbb{V}. e_2 \hookrightarrow v) \implies (\exists v \in \mathbb{V}. e_1 \hookrightarrow v)).$$

again, by the inclusion of evaluation in reduction, we can weaken:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies ((\exists v \in \mathbb{V}. e_2 \longrightarrow^* v) \implies (\exists v \in \mathbb{V}. e_1 \hookrightarrow v)).$$

Given the first assumption in this statement we can also say:

$$\forall e_1, e_2 \in \mathbb{E}. e_1 \longrightarrow e_2 \implies ((\exists v \in \mathbb{V}. e_1 \longrightarrow^* v) \implies (\exists v \in \mathbb{V}. e_1 \hookrightarrow v)),$$

and we no longer need the assumption as it is sufficient to show:

T2

$$\forall e_1 \in \mathbb{E}. ((\exists v \in \mathbb{V}. e_1 \longrightarrow^* v) \implies (\exists v \in \mathbb{V}. e_1 \hookrightarrow v)).$$

But note that T1 and T2 are identical goals. They state:

T

$$\forall e \in \mathbb{E}. ((\exists v \in \mathbb{V}. e \longrightarrow^* v) \implies (\exists v \in \mathbb{V}. e \hookrightarrow v)).$$

This statement is a direct consequence of Lemma 31. □

It is easy to show that $e \hookrightarrow v \implies e \longrightarrow^* v$, as it follows directly from Lemma 34.

6.2 Reduction is in Evaluation

Lemma 31 (Reduction is in Evaluation) $\forall e \in \mathbb{E}, v_1 \in \mathbb{V}.$

$$e \longrightarrow^* v_1 \implies (\exists v_3 \in \mathbb{V}. e \hookrightarrow v_3 \longrightarrow^* v_1).$$

Proof. We arrive at this result by an adaptation of Plotkin's proof for a similar result for the CBV and CBN lambda calculi [3]. The main steps in the development are:

1. We strengthen our goal to become:

$$e \longrightarrow^* v_1 \implies (\exists v_3 \in \mathbb{V}. e \hookrightarrow v_3 \longrightarrow^* v_1).$$

2. We define a *left reduction function* \mapsto (Section 2.11) such that (Lemma 36): $\forall e \in \mathbb{E}, v \in \mathbb{V}$.

$$e \mapsto^* v \iff e \hookrightarrow v$$

and $\forall e_1, e_2 \in \mathbb{E}. e_1 \mapsto e_2 \implies e_1 \longrightarrow e_2$ (Lemma 35). Thus, big-step evaluation (or simply evaluation) is exactly a chain of left reductions that ends in a value.

3. Our goal is restated as:

$$e \longrightarrow^* v_1 \implies (\exists v_3 \in \mathbb{V}. e \mapsto^* v_3 \longrightarrow^* v_1).$$

4. For technical reasons, the proofs are simpler if we use a parallel reduction relation \gg (Section 2.8) similar to the one introduced in the last section. Our goal is once again restated as:

$$e \gg^* v_1 \implies (\exists v_3 \in \mathbb{V}. e \mapsto^* v_3 \gg^* v_1).$$

5. The left reduction function induces a very fine classification $(\mathbb{V}, \mathbb{W}, \mathbb{S})$ on terms. In particular, any term $e \in \mathbb{E}$ must be exactly one of the following three (Lemma 20):

- (a) a *value* $e \in \mathbb{V}$,
- (b) a *workable* $e \in \mathbb{W}$, or
- (c) a *stuck* $e \in \mathbb{S}$,

where membership in each of these three sets is defined inductively over the structure of the term. We write v , w and s to refer to a member of one of the three sets above, respectively. Left reduction at level n is a total function exactly on the members of the set W^n (Lemma 22). Thus, left reduction is strictly undefined on non-workables, that is, it is undefined on values and on stuck terms. Furthermore, if the result of any parallel reduction is a value, the source must have been either a value or a workable (Lemma 21). We will refer to this property of parallel reduction as *monotonicity*.

6. Using the above classification, we break our goal into two cases, depending on whether the starting point is a value or a workable:

G1 $\forall v_1, v \in \mathbb{V}$.

$$v \gg^* v_1 \implies (\exists v_3 \in \mathbb{V}. v = v_3 \gg^* v_1),$$

G2 $\forall w \in \mathbb{W}, v \in \mathbb{V}$.

$$w \gg^* v_1 \implies (\exists v_3 \in \mathbb{V}. w \mapsto^+ v_3 \gg^* v_1).$$

It is obvious that G1 is true. Thus, G2 becomes the current goal.

7. By the monotonicity of parallel reduction, it is clear that all the intermediate terms in the reduction chain $w \gg^* v_1$ are either workables or values. Furthermore, workables and values do not interleave, and there is exactly one transition from workables to values in the chain. Thus, this chain can be visualised as follows:

$$w_1 \gg w_2 \gg \dots w_{k-1} \gg w_k \gg v \gg^* v_1.$$

We prove that the transition $w_k \gg v$ can be replaced by an evaluation (Lemma 37):

R1 $\forall w \in \mathbb{W}, v \in \mathbb{V}$.

$$w \gg v \implies (\exists v_2 \in \mathbb{V}. w \mapsto^+ v_2 \gg v).$$

With this lemma, we know that we can replace the chain above by one where the evaluation involved in going from the last workable to the first value is explicit:

$$w_1 \gg w_2 \gg \dots w_{k-1} \gg w_k \mapsto^+ v_2 \gg^* v_1.$$

What is left is then to “push back” this information about the last workable in the chain to the very first workable in the chain. This is achieved by a straightforward iteration (by induction over the number of k of workables in the chain) of a result that we prove (Lemma 32):

R2 $\forall w_1, w_2 \in \mathbb{W}, v_1 \in \mathbb{V}$.

$$w_1 \gg w_2 \mapsto^+ v_1 \implies (\exists v_2 \in \mathbb{V}. w_1 \mapsto^+ v_2 \gg v_1).$$

With this result, we are able to move the predicate $_ \mapsto^+ v_3 \gg^* v$ all the way back to the first workable in the chain. This step can be visualised as follows. With one application of R2 we have the chain:

$$w_1 \gg w_2 \gg \dots w_{k-1} \mapsto^+ v_3 \gg^* v_1,$$

and with $k - 2$ applications of R2 we have:

$$w_1 \mapsto^+ v_{k+1} \gg^* v_1,$$

thus completing the proof. □

6.3 Push Back

Lemma 32 (Push Back) $\forall X \in \mathbb{N}, w_1, w_2 \in W, v_2 \in \mathbb{V}.$

$$w_1 \gg^X w_2 \mapsto^+ v_1 \implies (\exists v_2 \in \mathbb{V}. w_1 \mapsto^+ v_2 \gg v_1).$$

Proof. The assumption corresponds to a chain of reductions:

$$w_1 \gg w_2 \mapsto w_3 \mapsto \dots w_{k-1} \mapsto w_k \mapsto v_1.$$

Applying Permutation to $w_1 \gg w_2 \mapsto w_3$ gives us $(\exists e_{2'} \in \mathbb{E}. w_1 \mapsto^+ e_{2'} \gg w_3)$. By the monotonicity of parallel reduction, we know that only a workable can reduce to a workable, that is, $(\exists w_{2'} \in W^n. w_1 \mapsto^+ w_{2'} \gg w_3)$. Now we have the chain:

$$w_1 \mapsto^+ w_{2'} \gg w_3 \mapsto \dots w_{k-1} \mapsto w_k \mapsto v_1.$$

Repeating this step $k - 2$ times we have:

$$w_1 \mapsto^+ w_{2'} \mapsto^+ w_{3'} \mapsto^+ \dots w_{k-1'} \gg w_k \mapsto v_1.$$

Applying Permutation to $w_{k-1'} \gg w_k \mapsto v_1$ give us $(\exists e_{k'} \in \mathbb{E}. w_{k-1'} \mapsto^+ e_{k'} \gg v_1)$. By the monotonicity of parallel reduction, we know that $e_{k'}$ can only be a value or a workable. If it is a value then we have the chain:

$$w_1 \mapsto^+ w_{2'} \mapsto^+ w_{3'} \mapsto^+ \dots w_{k-1'} \mapsto^+ v_{k'} \gg v_1$$

and we are done. If it is a workable, then applying Transition to $w_{k'} \gg v_1$ gives us $(\exists v_2 \in V. w_{k'} \mapsto^+ v_2 \gg v_1)$. This means that we now have the chain:

$$w_1 \mapsto^+ w_{2'} \mapsto^+ w_{3'} \mapsto^+ \dots w_{k-1'} \mapsto^+ w_{k'} \mapsto^+ v_2 \gg v_1$$

and we are done. □

7 Special Part of Computational Adequacy of Reduction Semantics

Remark 33 (Non-termination and the Finiteness of Trees) *The reader should be reminded here that all the structures that we ever construct in this development are finite trees. Thus, non-terminating computations are not modelled by infinite derivations, but rather, by the absence of a “conclusive” derivation. In particular, a big-step derivation is simply absent for a “non-terminating” computation, and thus, the big-step semantics identifies stuck and non-terminating computations. Similarly, a small-step derivation is always defined on a non-terminating computation, but every finite sequence of small-step can be extended by another step. Thus, no finite sequence of small-steps leads to a value (or a stuck for that matter). Note, however, that the small-step semantics allows us to distinguish between a stuck (which cannot be advanced by small-step reduction) and a workable (which can be advanced an arbitrarily large number of times by small-step reduction).*

7.1 Evaluation is in Reduction

Lemma 34 (Evaluation is in Reduction) $\forall e \in \mathbb{E}, v \in \mathbb{V}$.

$$e \hookrightarrow v \implies e \longrightarrow^* v.$$

Proof. By a straightforward induction on the height of the judgement $e \hookrightarrow v$.

1. $() \hookrightarrow ()$ and $() \longrightarrow^* ()$.
2. $\lambda x.e \hookrightarrow \lambda x.e$ and $\lambda x.e \longrightarrow^* \lambda x.e$
3. $\#z \hookrightarrow \#z$. obviously $\#z \longrightarrow^* \#z$
4. $\lambda(\#z \Rightarrow x).e \hookrightarrow \lambda(\#z \Rightarrow x).e$ and $\lambda(\#z \Rightarrow x).e \longrightarrow^* \lambda(\#z \Rightarrow x).e$.
5. $\lambda^{i \in L} f x_i.e_i \hookrightarrow \lambda^{i \in L} f x_i.e_i$ and $\lambda^{i \in L} f x_i.e_i \longrightarrow^* \lambda^{i \in L} f x_i.e_i$

$$\begin{array}{c} e_1 \hookrightarrow \lambda x.e \\ e_2 \hookrightarrow e_3 \\ \hline e[x := e_3] \hookrightarrow e_4 \end{array}$$

6. By induction hypothesis, $e_1 \longrightarrow^* \lambda x.e$, $e_2 \longrightarrow^* e_3$ and $e[x := e_3] \longrightarrow^* e_4$. Then, by compatibility of \longrightarrow^* :
 $e_1 e_2 \longrightarrow^* (\lambda x.e) e_2 \longrightarrow^* (\lambda x.e) e_3 \longrightarrow_{\beta_1} e[x := e_3] \longrightarrow^* e_4$.

$$\begin{array}{c} e_1 \hookrightarrow \lambda(\#z \Rightarrow x).e \\ e_2 \hookrightarrow \#z \Rightarrow b_3 \\ \hline e[x := \lambda x'.(b_3[\#z := x'])] \hookrightarrow e_4 \\ \hookrightarrow \uparrow \frac{e_1 e_2 \hookrightarrow e_4}{IH \downarrow \quad IH \downarrow \quad IH \downarrow} \\ \begin{array}{c} e_1 \longrightarrow^* \lambda(\#z \Rightarrow x).e \\ e_2 \longrightarrow^* \#z \Rightarrow b_3 \\ e[x := \lambda x'.(b_3[\#z := x'])] \longrightarrow^* e_4 \end{array} \\ \hline e_1 e_2 \longrightarrow^* (\lambda(\#z \Rightarrow x).e) e_2 \longrightarrow^* (\lambda(\#z \Rightarrow x).e) (\#z \Rightarrow b_3) \longrightarrow_{\beta_2} e[x := \lambda x.b_3[\#z := x']] \longrightarrow^* e_4 \text{ comp. } \longrightarrow^* \downarrow \end{array}$$

$$\begin{array}{c} e_1 \hookrightarrow \lambda^{i \in L \cup \{k\}} f_i x_i.e_i \\ e_2 \hookrightarrow f_k e_4 \quad k \leq n \\ e_k[x := e_4] \hookrightarrow e_5 \\ \hline \hookrightarrow \uparrow \frac{e_1 e_2 \hookrightarrow e_5}{IH \downarrow \quad IH \downarrow \quad IH \downarrow} \\ \begin{array}{c} e_1 \longrightarrow^* \lambda^{i \in L \cup \{k\}} f_i x_i.e_i \\ e_2 \longrightarrow^* f_k e_4 \quad k \leq n \\ e_k[x := e_4] \longrightarrow^* e_5 \end{array} \\ \hline e_1 e_2 \longrightarrow^* (\lambda^{i \in F \cup \{k\}} f_i x_i.e_i) e_2 \longrightarrow^* (\lambda^{i \in F \cup \{k\}} f_i x_i.e_i) (f_k e_4) \longrightarrow_{\beta_3} e_k[x := e_4] \longrightarrow^* e_5 \text{ comp. } \longrightarrow^* \downarrow \end{array}$$

$$\begin{array}{c} e_1 \hookrightarrow v_1 \quad \xRightarrow{IH} \quad e_1 \longrightarrow^* v_1 \\ e_2 \hookrightarrow v_2 \quad \xRightarrow{IH} \quad e_2 \longrightarrow^* v_2 \\ \hline \hookrightarrow \uparrow \frac{(e_1, e_2) \hookrightarrow (v_1, v_2)}{(e_1, e_2) \longrightarrow^* (v_1, e_2) \longrightarrow^* (v_1, v_2)} \text{ comp. } \longrightarrow^* \downarrow \end{array}$$

$$\begin{array}{c} e \hookrightarrow (v_1, v_2) \quad \xRightarrow{IH} \quad e \longrightarrow^* (v_1, v_2) \\ \pi_1 e \hookrightarrow v_1 \quad \xRightarrow{IH} \quad \pi_1 e \longrightarrow^* \pi_1 (v_1, v_2) \longrightarrow_{\pi_1} v_1 \\ \hline \hookrightarrow \uparrow \frac{e \hookrightarrow (v_1, v_2)}{\pi_1 e \hookrightarrow v_1} \text{ comp. } \longrightarrow^* \downarrow \end{array}$$

$$\begin{array}{c} e \hookrightarrow (v_1, v_2) \quad \xRightarrow{IH} \quad e \longrightarrow^* (v_1, v_2) \\ \pi_2 e \hookrightarrow v_2 \quad \xRightarrow{IH} \quad \pi_2 e \longrightarrow^* \pi_2 (v_1, v_2) \longrightarrow_{\pi_2} v_2 \\ \hline \hookrightarrow \uparrow \frac{e \hookrightarrow (v_1, v_2)}{\pi_2 e \hookrightarrow v_2} \text{ comp. } \longrightarrow^* \downarrow \end{array}$$

$$\begin{array}{c} e_1 \hookrightarrow e_2 \quad \xRightarrow{IH} \quad e_1 \longrightarrow^* e_2 \\ f_k e_1 \hookrightarrow f_k e_2 \quad \xRightarrow{IH} \quad f_k e_1 \longrightarrow^* f_k e_2 \\ \hline \hookrightarrow \uparrow \frac{e_1 \hookrightarrow e_2}{f_k e_1 \hookrightarrow f_k e_2} \text{ comp. } \longrightarrow^* \downarrow \end{array}$$

$$\begin{array}{c} e_1 \hookrightarrow v \quad \xRightarrow{IH} \quad e_1 \longrightarrow^* v \\ \#z \Rightarrow e_1 \hookrightarrow \#z \Rightarrow v \quad \xRightarrow{IH} \quad \#z \Rightarrow e_1 \longrightarrow^* \#z \Rightarrow v \\ \hline \hookrightarrow \uparrow \frac{e_1 \hookrightarrow v}{\#z \Rightarrow e_1 \hookrightarrow \#z \Rightarrow v} \text{ comp. } \longrightarrow^* \downarrow \end{array}$$

$$\begin{array}{c} e_1 \hookrightarrow \#z \quad \xRightarrow{IH} \quad e_1 \longrightarrow^* \#z \\ e_2 \hookrightarrow \#z \quad \xRightarrow{IH} \quad e_2 \longrightarrow^* \#z \\ \hline \hookrightarrow \uparrow \frac{e_1 \hookrightarrow \#z \quad e_2 \hookrightarrow \#z}{e_1 =_{\#} e_2 \hookrightarrow \text{True}()} \text{ comp. } \longrightarrow^* \downarrow \end{array}$$

$$\begin{aligned}
15. \hookrightarrow \uparrow & \frac{\frac{e_1 \hookrightarrow \#z}{e_2 \hookrightarrow \#z'} \quad \#z \neq \#z'}{e_1 = e_2 \hookrightarrow \text{False}()} \xrightarrow{IH} \frac{\frac{e_1 \rightarrow^* \#z}{e_2 \rightarrow^* \#z'}}{e_1 =_{\#} e_2 \rightarrow^* \#z = e_2 \rightarrow^* \#z = \#z' \rightarrow_{\#} \text{False}()} \text{comp.} \rightarrow^* \Downarrow \\
16. \hookrightarrow \uparrow & \frac{e \hookrightarrow \#z}{\text{isOVar } e \hookrightarrow \text{True}()} \xrightarrow{IH} \frac{e \rightarrow^* \#z}{\text{isOVar } e \rightarrow^* \text{isOVar } \#z \rightarrow_{\delta_{\text{isOVar}}} \text{True}()} \text{comp.} \rightarrow^* \Downarrow \\
17. \hookrightarrow \uparrow & \frac{e \hookrightarrow v \quad v \neq \#z}{\text{isOVar } e \hookrightarrow \text{False}()} \xrightarrow{IH} \frac{e \rightarrow^* v \quad v \neq \#z}{\text{isOVar } e \rightarrow^* \text{isOVar } v \rightarrow_{\delta_{\text{isOVar}}} \text{False}()} \text{comp.} \rightarrow^* \Downarrow
\end{aligned}$$

What is harder to show is the “converse”, that is, that $e \rightarrow^* v \implies (\exists v' \in V. e \hookrightarrow v')$. It is a consequence of the stronger result of Lemma 31.

In the rest of this section, we present the definitions and lemmas mentioned above.

7.2 Left Reduction is in Reduction

Lemma 35 (Left Reduction is in Reduction) $\forall e_1, e_2 \in \mathbb{E}$.

$$e_1 \mapsto e_2 \implies e_1 \rightarrow e_2.$$

Proof (Lemma 35). Proof is straightforward, by induction on the height of the first judgement.

1. If $(\lambda x. e) v \mapsto e[x := v]$, and $(\lambda x. e) v \rightarrow_{\beta_1} e[x := v]$.
2. $\hookrightarrow \uparrow \frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \xrightarrow{IH} \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \text{comp.} \rightarrow \Downarrow$
3. $\hookrightarrow \uparrow \frac{e \mapsto e'}{v e \mapsto v e'} \xrightarrow{IH} \frac{e \rightarrow e'}{v e \rightarrow v e'} \text{comp.} \rightarrow \Downarrow$
4. $\hookrightarrow \uparrow \frac{e_1 \mapsto e'_1}{(e_1, e_2) \mapsto (e'_1, e_2)} \xrightarrow{IH} \frac{e_1 \rightarrow e'_1}{(e_1, e_2) \rightarrow (e'_1, e_2)} \text{comp.} \rightarrow \Downarrow$
5. $\hookrightarrow \uparrow \frac{e \mapsto e'}{(v, e) \mapsto (v, e')} \xrightarrow{IH} \frac{e \rightarrow e'}{(v, e) \rightarrow (v, e')} \text{comp.} \rightarrow \Downarrow$
6. $\hookrightarrow \uparrow \frac{e \mapsto e'}{\pi_n e \mapsto \pi_n e'} \xrightarrow{IH} \frac{e \rightarrow e'}{\pi_n e \rightarrow \pi_n e'} \text{comp.} \rightarrow \Downarrow$
7. $\pi_1 (v_1, v_2) \mapsto v_1$, and $\pi_1 (v_1, v_2) \rightarrow_{\pi} v_1$
8. $\pi_2 (v_1, v_2) \mapsto v_2$, and $\pi_2 (v_1, v_2) \rightarrow_{\pi} v_2$
9. $(\lambda^{i \in L \cup \{k\}} f_i x_i. e_i) (f_k v) \mapsto e_k[x_k := v]$ and $(\lambda^{i \in L \cup \{k\}} f_i x_i. e_i) (f_k v) \rightarrow_{\beta_3} e_k[x_k := v]$.
10. $\hookrightarrow \uparrow \frac{e \mapsto e'}{f e \mapsto f e'} \xrightarrow{IH} \frac{e \rightarrow e'}{f e \rightarrow f e'} \text{comp.} \rightarrow \Downarrow$
11. $(\lambda(\#z \Rightarrow x). e) (\#z' \Rightarrow b) \mapsto e[x := \lambda x'. b[\#z' := x']]$, and $(\lambda(\#z \Rightarrow x). e) (\#z' \Rightarrow b) \rightarrow_{\beta_2} e[x := \lambda x'. b[\#z' := x']]$.
12. $\hookrightarrow \uparrow \frac{e \mapsto e'}{\#z \Rightarrow e \mapsto \#z \Rightarrow e'} \xrightarrow{IH} \frac{e \rightarrow e'}{\#z \Rightarrow e \rightarrow \#z \Rightarrow e'} \text{comp.} \rightarrow \Downarrow$
13. $\#z =_{\#} z \mapsto \text{True}()$, and $\#z = \#z \rightarrow_{\#} \text{True}()$
14. $\frac{\#z \neq \#z'}{\#z =_{\#} z' \mapsto \text{False}()}$ and $\frac{\#z \neq \#z'}{\#z = \#z' \rightarrow_{\#} \text{False}()}$.
15. $\hookrightarrow \uparrow \frac{e_1 \mapsto e'_1}{e_1 =_{\#} e_2 \mapsto e'_1 =_{\#} e_2} \xrightarrow{IH} \frac{e_1 \rightarrow e'_1}{e_1 = e_2 \rightarrow e'_1 = e_2} \text{comp.} \rightarrow \Downarrow$
16. $\hookrightarrow \uparrow \frac{e \mapsto e'}{\#z' =_{\#} e \mapsto \#z' =_{\#} e'} \xrightarrow{IH} \frac{e \rightarrow e'}{\#z' = e \rightarrow \#z' = e'} \text{comp.} \rightarrow \Downarrow$
17. $\text{isOVar } \#z \mapsto \text{True}()$ and $\text{isOVar } \#z \rightarrow_{\delta_{\text{isOVar}}} \text{True}()$
18. If $v \neq \#z$, $\text{isOVar } v \mapsto \text{True}()$ and $\text{isOVar } v \rightarrow_{\delta_{\text{isOVar}}} \text{True}()$
19. $\hookrightarrow \uparrow \frac{e \mapsto e'}{\text{isOVar } e \mapsto \text{isOVar } e'} \xrightarrow{IH} \frac{e \rightarrow e'}{\text{isOVar } e \rightarrow \text{isOVar } e'} \text{comp.} \rightarrow \Downarrow$

□

7.3 Left Reduction is Evaluation

Lemma 36 (Left Reduction and Big-step Semantics) $\forall e \in \mathbb{E}, v \in \mathbb{V}.$

$$e \mapsto^* v \iff e \hookrightarrow v.$$

Proof. The forward direction: $\forall e \in \mathbb{E}, v \in \mathbb{V}. e \mapsto^* v \implies e \hookrightarrow v$

By induction on the length k of derivation $e \mapsto^k$, and then by induction on the size of e . It proceeds by case analysis on e .

– $k = 0$

1. For all values v , $v \mapsto^* v$, since \mapsto^* is reflexive. For all other expressions $e \mapsto^* e$, e is not a value, and the property holds by contradiction.

– $k = n + 1$.

1. $e_1 e_2 \mapsto^i (\lambda x.e) e_2 \mapsto^j (\lambda x.e) v' \mapsto^1 e[x := v'] \mapsto^k v$ where $i + j + k = n$. Then, by induction hypothesis

$$\frac{\begin{array}{c} e_1 \hookrightarrow (\lambda x.e) \\ e_2 \hookrightarrow v' \\ e[x := v'] \hookrightarrow v \end{array}}{e_1 e_2 \hookrightarrow v} \hookrightarrow \Downarrow$$

2. $e_1 e_2 \mapsto^i (\lambda(\#_ \Rightarrow x).e) e_2 \mapsto^j (\lambda x.e) (\#z \Rightarrow b') \mapsto^1 e[x := \lambda x'.b'[\#z := x']] \mapsto^k v$ where $i + j + k = n$. Then, by induction hypothesis

$$\frac{\begin{array}{c} e_1 \hookrightarrow \lambda(\#_ \Rightarrow x).e \\ e_2 \hookrightarrow \#z \Rightarrow b' \\ e[x := \lambda x'.b'[\#z := x']] \hookrightarrow v \end{array}}{e_1 e_2 \hookrightarrow v} \hookrightarrow \Downarrow$$

3. $e_1 e_2 \mapsto^i (\lambda^{l \in F \cup \{o\}} f_l x_l.e_l) e_2 \mapsto^j (\lambda^{l \in F \cup \{o\}} f_l x_l.e_l) (f_o v') \mapsto^1 e_o[x := v'] \mapsto^k v$ where $i + j + k = n$. Then, by induction hypothesis

$$\frac{\begin{array}{c} e_1 \hookrightarrow (\lambda^{l \in F \cup \{o\}} f_l x_l.e_l) \\ e_2 \hookrightarrow f_o v' \\ e_l[x := v'] \hookrightarrow v \end{array}}{e_1 e_2 \hookrightarrow v} \hookrightarrow \Downarrow$$

4. $(e_1, e_2) \mapsto^i (v_1, e_2) \mapsto^j (v_1, v_2)$, where $i + j = n + 1$. Then, by the induction hypothesis

$$\frac{e_1 \hookrightarrow v_1 \quad e_2 \hookrightarrow v_2}{(e_1, e_2) \hookrightarrow (v_1, v_2)} \hookrightarrow \Downarrow$$

5. $\pi_1 e \mapsto^n \pi_1 (v_1, v_2) \mapsto^1 v_1$. For each step in the n first reductions, clearly the same rule of the left reduction applies, namely $\frac{e \mapsto e'}{\pi_1 e \mapsto \pi_1 e'}$. Thus it easily follows that $e \mapsto^n (v_1, v_2)$. Applying the induction hypothesis to this result, we obtain $e \hookrightarrow (v_1, v_2)$. By definition of \hookrightarrow , from this follows $\pi_1 e \hookrightarrow v_1$.

6. $\pi_2 e \mapsto^n \pi_2 (v_1, v_2) \mapsto^1 v_2$. For each step in the n first reductions, clearly the same rule of the left reduction applies, namely $\frac{e \mapsto e'}{\pi_2 e \mapsto \pi_2 e'}$. Thus it easily follows that $e \mapsto^n (v_1, v_2)$. Applying the induction hypothesis to this result, we obtain $e \hookrightarrow (v_1, v_2)$. By definition of \hookrightarrow , from this follows $\pi_2 e \hookrightarrow v_2$.

7.

$$\frac{e_1 \mapsto^{n+1} v}{f e_1 \mapsto^{n+1} f v} \xrightarrow{IH} \frac{e_1 \hookrightarrow v}{f e_1 \hookrightarrow f v}$$

8.

$$\frac{e_1 \mapsto^{n+1} v}{\#z \Rightarrow e_1 \mapsto^{n+1} \#z \Rightarrow v} \xrightarrow{IH} \frac{e_1 \hookrightarrow v}{\#z \Rightarrow e_1 \hookrightarrow \#z \Rightarrow v} \hookrightarrow \Downarrow$$

9. $e_1 = e_2 \mapsto^i \#z = e_2 \mapsto^j \#z = \#z \mapsto^1 \text{True}$, where $i + j = n$. Then, by the induction hypothesis

$$\frac{\begin{array}{c} e_1 \hookrightarrow \#z \\ e_2 \hookrightarrow \#z \end{array}}{e_1 = e_2 \hookrightarrow \text{True}()} \hookrightarrow \Downarrow$$

10. Similarly,

$e_1 = e_2 \mapsto^i \#z = e_2 \mapsto^j \#z = \#z' \mapsto^1 \text{False}$, where $i + j = n$. Then, by the induction hypothesis

$$\frac{\begin{array}{c} e_1 \hookrightarrow \#z \\ e_2 \hookrightarrow \#z' \end{array}}{e_1 = e_2 \hookrightarrow \text{False}()} \hookrightarrow \Downarrow$$

11. $\text{isOVar } e \mapsto^i \text{isOVar } \#z \mapsto^1 \text{True}()$, then $e \mapsto^i \#z$. By the induction hypothesis $e \hookrightarrow \#z$. Then, $\text{isOVar } e \hookrightarrow \text{True}()$.

12. $\text{isOVar } e \mapsto^i \text{isOVar } v \mapsto^1 \text{False}()$, then $e \mapsto^i v$. By the induction hypothesis $e \hookrightarrow v$. Then, $\text{isOVar } e \hookrightarrow \text{False}()$.

The backward direction: $\forall e \in \mathbb{E}. \forall v \in \mathbb{V}. e \hookrightarrow v \implies e \mapsto^* v$. By induction on the height of the derivation of $e \hookrightarrow v$, and then by the size of e .

1. $\frac{}{() \hookrightarrow ()}$ and $() \mapsto^0 ()$.

2. $\frac{}{\lambda x.e \hookrightarrow \lambda x.e}$ and $\lambda x.e \mapsto^0 \lambda x.e$.

3. $\frac{e[x:=e_3] \hookrightarrow e_4}{e_1 e_2 \hookrightarrow e_4}$ By induction hypothesis: $e_1 \mapsto^i (\lambda x.e)$ $e_2 \mapsto^j v$. Then, $e_1 e_2 \mapsto^i (\lambda x.e) e_2 \mapsto^j (\lambda x.e) v \mapsto^1 e[x:=e_3] \mapsto^k e_4$.

4. $\frac{e[x:=v] \mapsto^k e_4}{e_1 \hookrightarrow \lambda^{f \in L \cup \{k\}}(f x_f).e_f}$ $e_2 \mapsto^j k e_4$ $e_k[x:=e_4] \mapsto^l e_5$. By induction hypothesis: $e_1 \mapsto^i \lambda^{i \in L \cup \{k\}}(f_i x_i).e_i$ $e_2 \mapsto^j k e_4$ $e_k[x:=e_4] \mapsto^l e_5$. Then, $e_1 e_2 \mapsto^i (\lambda^{i \in L \cup \{k\}}(f_i x_i).e_i) e_2 \mapsto^j k e_4 \mapsto^l e_5$.

5. $\frac{e[x:=\lambda x'.(b_3[\#z:=x'])] \hookrightarrow e_4}{e_1 e_2 \hookrightarrow e_4}$. By the induction hypothesis $e_1 \mapsto^i \lambda(\#_ \Rightarrow x).e$ $e_2 \mapsto^j \#z \Rightarrow b_3$ $e[x:=\lambda x'.(b_3[\#z:=x'])] \mapsto^k e_4$. Then, $e_1 e_2 \mapsto^i (\lambda(\#_ \Rightarrow x).e) e_2 \mapsto^j (\lambda(\#_ \Rightarrow x).e) (\#z \Rightarrow b_3) \mapsto^1 e[x:=\lambda x'.b[\#z:=x']] \mapsto^k e_4$.

6. $\frac{e_1 \hookrightarrow e_3 \quad e_2 \hookrightarrow e_4}{(e_1, e_2) \hookrightarrow (e_3, e_4)}$ By the induction hypothesis $e_1 \mapsto^i e_3$ $e_2 \mapsto^j e_4$. Then, $(e_1, e_2) \mapsto^i (e_3, e_2) \mapsto^j (e_3, e_4)$.

7. $\frac{e \hookrightarrow (e_3, e_4)}{\pi_1 e \hookrightarrow e_3}$. By the induction hypothesis: $e \mapsto^n (e_3, e_4)$. Then, $\pi_1 e \mapsto^n \pi_1 (e_3, e_4) \mapsto^1 e_3$.

8. $\frac{e \hookrightarrow (e_3, e_4)}{\pi_2 e \hookrightarrow e_4}$. By the induction hypothesis: $e \mapsto^n (e_3, e_4)$. Then, $\pi_2 e \mapsto^n \pi_2 (e_3, e_4) \mapsto^1 e_4$.

9. $\frac{e_1 \hookrightarrow e_2}{f_k e_1 \hookrightarrow f_k e_2}$ By the induction hypothesis $e_1 \mapsto^{n+1} e_2$. Then, $f e_1 \mapsto^{n+1} f e_2$.

10. $\frac{}{\lambda^{i \in L} f_i x_i.e_i \hookrightarrow \lambda^{i \in L} f_i x_i.e_i}$ and $\lambda^{i \in L} f_i x_i.e_i \mapsto^0 \lambda^{i \in L} f_i x_i.e_i$.

11. $\frac{e \hookrightarrow v}{\#z \Rightarrow e \hookrightarrow \#z \Rightarrow v}$ By the induction hypothesis $e \mapsto^{n+1} v$. Then, $\#z \Rightarrow e \mapsto^{n+1} \#z \Rightarrow v$.

12. $\#z \hookrightarrow \#z$, and $\#z \mapsto^0 \#z$.

13. $\lambda(z \Rightarrow x).e \hookrightarrow \lambda(z \Rightarrow x).e$, and $\lambda(z \Rightarrow x).e \mapsto^0 \lambda(z \Rightarrow x).e$.
14. $\frac{e_1 \hookrightarrow \#z \quad e_2 \hookrightarrow \#z}{e_1 = \# \quad e_2 \hookrightarrow \text{True}() \quad \#z \mapsto^1 \text{True}()}. \text{ By the induction hypothesis: } \frac{e_1 \mapsto^i \#z \quad e_2 \mapsto^j \#z}{e_1 = \# \quad e_2 \mapsto^i \#z = \# \quad e_2 \mapsto^j \#z = \#}$
15. $\frac{e_1 \hookrightarrow \#z \quad e_2 \hookrightarrow \#z' \quad \#z' \neq \#z}{e_1 = \# \quad e_2 \hookrightarrow \text{True}() \quad \#z' \mapsto^1 \text{False}()}. \text{ By the induction hypothesis: } \frac{e_1 \mapsto^i \#z \quad e_2 \mapsto^j \#z'}{e_1 = \# \quad e_2 \mapsto^i \#z = \# \quad e_2 \mapsto^j \#z = \#}$
16. $\text{Rulee} \hookrightarrow \#z \text{isOVar } e \hookrightarrow \text{True}() \text{ By the induction hypothesis: } e \mapsto^* \#z. \text{ Then, isOVar } e \mapsto^* \text{isOVar } \#z \mapsto \text{True}().$
17. $\frac{e \hookrightarrow v \quad v \neq \#z}{\text{isOVar } e \hookrightarrow \text{False}()}. \text{ By the induction hypothesis: } e \mapsto^* v \text{ (and } v \neq \#z). \text{ Then, isOVar } e \mapsto^* \text{isOVar } v \mapsto \text{False}().$

7.4 Transition Lemma

Lemma 37 (Transition) $\forall X \in \mathbb{N}. \forall w \in W, v \in \mathbb{V}.$

$$w \gg^X v \implies \exists v_2 \in \mathbb{V}, Y \in \mathbb{N}. w \mapsto^+ v_2 \gg v \wedge Y < X.$$

Proof (Lemma 37(Transition)). Proof is by induction on the complexity X , and then on the structure of w .

1. For the workable $(\lambda x.e) v$, $\frac{e \gg^M e' \quad v \gg^N v'}{(\lambda x.e) v \gg^{M+\#(x,e')N+1} e'[x:=v']}$

Since, $e \gg^M e'$ and $v \gg^N v'$, then by lemma 18, $\exists Z. e[x:=v] \gg^Z e'[x:=v'] \wedge Z \leq M + \#(x,e')N$. Now, there are two possibilities.

- (a) First, if $e[x:=v]$ is a workable, then since $Z < M + \#(x,e')N + 1$, the induction hypothesis applies to $e[x:=v]$, and therefore: $\exists v_1. \exists Y_1. e[x:=v] \mapsto^+ v_1 \gg^Y e'[x:=v'] \wedge Y_1 < Z$. Thus, we obtain our goal as follows: $\exists v_2 = v_1. \exists Y = Y_1.$

$$(\lambda x.e) v \mapsto^1 e[x:=v] \mapsto^+ v_2 \gg^Y e'[x:=v'] \wedge Y < M + \#(x,e')N + 1$$

- (b) Otherwise, if $e[x:=v]$ is a value, then $\exists v_2 = e[x:=v]. \exists Y = Z.$

$$(\lambda x.e) v \mapsto^1 e[x:=v] \gg^Y e'[x:=v'] \wedge Y < M + \#(x,e')N + 1$$

2. For the workable $(\lambda(\#_ \Rightarrow x).e) (\#z \Rightarrow b)$ $\frac{e \gg^M e'}{(\lambda(\#_ \Rightarrow x).e) (\#z \Rightarrow b) \gg^{M+1} e'[x:=\lambda x'.b[\#z:=x']]}$. Since $e \gg^M e'$

and $\lambda x'.b[\#z:=x'] \gg^0 \lambda x'.b[\#z:=x']$, by lemma 18 $\exists Z. e[x:=\lambda x'.b[\#z:=x']] \gg^Z e'[x:=\lambda x'.b[\#z:=x']] \wedge Z \leq M$. Again, there are two possibilities

- (a) $e[x:=\lambda x'.b[\#z:=x']]$ is a workable. Then, since $Z < M + 1$, the induction hypothesis applies and we obtain $\exists v_1. \exists Y_1. e[x:=\lambda x'.b[\#z:=x']] \mapsto^+ v_1 \gg^{Y_1} e'[x:=\lambda x'.b[\#z:=x']] \wedge Y_1 < Z$. Then, it easily follows $\exists v_2 = v_1. \exists Y = Y_1.$

$$(\lambda(\#_ \Rightarrow x).e) (\#z \Rightarrow b) \mapsto^1 e[x:=\lambda x'.b[\#z:=x']] \mapsto^+ v_2 \gg^Y e'[x:=\lambda x'.b[\#z:=x']] \wedge Y < M + 1$$

- (b) Otherwise, $e[x:=\lambda x'.b[\#z:=x']]$ is a value. Then, $\exists v_2 = e[x:=\lambda x'.b[\#z:=x']]. \exists Y = Z.$

$$(\lambda(\#_ \Rightarrow x).e) (\#z \Rightarrow b) \mapsto^1 e[x:=\lambda x'.b[\#z:=x']] \gg^Y e'[x:=\lambda x'.b[\#z:=x']] \wedge Y < M + 1$$

3. The workable $w \ e$ can never parallel-reduce to a value in one parallel reduction step, so the property vacuously holds.
4. Similarly, the workable $v \ w$ can never parallel-reduce to a value in one parallel reduction step, so the property vacuously holds.
5. For the workable $f \ w$, $\frac{w \gg^X v}{f \ w \gg^X v}$.

By the induction hypothesis $\exists v_1. \exists Y_1. w \mapsto^+ v_1 \gg^{Y_1} v \wedge Y_1 < X$. Then, it easily follows that $\exists v_2 = f \ v_1. \exists Y = Y_1$.

$$f \ w \mapsto^+ f \ v_2 \gg^Y f \ v \wedge Y < X$$

6. For the workable $\#z \Rightarrow w$, $\frac{w \gg^M v}{\#z \Rightarrow w \gg^M \#z \Rightarrow v}$.

Then, by the induction hypothesis, $\exists v_1. \exists Y_1. w \mapsto^+ v_1 \gg^{Y_1} v \wedge Y_1 \leq M$.

Then, it easily follows that $\exists v_2 = \#z \Rightarrow v_1. \exists Y = Y_1. (\#z \Rightarrow w) \mapsto^+ v_2 \gg^Y (\#z \Rightarrow v) \wedge Y < M$.

7. For the workable (w, e) , $\frac{w \gg^M v_1 \quad e \gg^N v_2}{(w, e) \gg^{M+N} (v_1, v_2)}$.

Then, by the induction hypothesis $\exists v_3. \exists Y_1. w \mapsto^+ v_3 \gg^{Y_1} v_1 \wedge Y_1 < M$.

From this it easily follows: $\exists v_4 = (v_3, e). \exists Y = Y_1 + N$.

$$(w, e) \mapsto^+ (v_3, e) \gg^Y (v_1, v_2) \wedge Y < M + N$$

8. For the workable (v, w) , $\frac{v \gg^M v_1 \quad w \gg^N v_2}{(v, w) \gg^{M+N} (v_1, v_2)}$.

Then, by the induction hypothesis: $\exists v_3. \exists Y_1. w \mapsto^+ v_3 \gg^{Y_1} v_2 \wedge Y < N$.

From this it easily follows: $\exists v_3 = (v, v_3). \exists Y = M + Y_1$.

$$(v, w) \mapsto^+ (v, v_3) \gg^Y (v_1, v_2) \wedge Y < M + N$$

9. For the workable $w = e$, it is easy to show that it can never be parallel-reduced in one steps to a value, so the property holds vacuously.
10. Similarly for the workable $\#z = w$ it is easy to show that it can never be parallel-reduced in one step to a value, so the property holds vacuously.
11. For the workable $\#z = \#z'$, $\#z = \#z' \gg^1 \text{True}()$ if $\#z = \#z'$. Then, obviously, $\exists v_2 = \text{True}(). \exists Y = 0. \#z = \#z' \mapsto^1 \text{True}() \gg^0 \text{True}() \wedge 0 < 1$.
12. For the workable $\#z = \#z'$, $\#z = \#z' \gg^1 \text{False}()$ if $\#z \neq \#z'$. Then, obviously, $\exists v_2 = \text{False}(). \exists Y = 0. \#z = \#z' \mapsto^1 \text{False}() \gg^0 \text{False}() \wedge 0 < 1$.
13. For the workable $\pi_1 \ w$, it is easy to show that they could never reduced in one step to values by a single parallel reduction step, so the property vacuously holds.
14. For the workable $\pi_2 \ w$ it is easy to show that they could never reduced in one step to values by a single parallel reduction step, so the property vacuously holds.
15. For the workable $\pi_1 \ (v_1, v_2)$, $\frac{v_1 \gg^M v'_1}{\pi_1 \ (v_1, v_2) \gg^{M+1} v'_1}$. Then, $\exists v_2 = v_1. \exists Y = M. \pi_1 \ (v_1, v_2) \mapsto^1 v_1 \gg^M v'_1 \wedge Y < M+1$.
16. For the workable $\pi_2 \ (v_1, v_2)$, $\frac{v_2 \gg^M v'_2}{\pi_2 \ (v_1, v_2) \gg^{M+1} v'_2}$. Then, $\exists v_2 = v_2. \exists Y = M. \pi_2 \ (v_1, v_2) \mapsto^1 v_2 \gg^M v'_2 \wedge Y < M+1$.
17. For the workable $\text{isOVar} \ \#z$, $\text{isOVar} \ \#z \gg^1 \text{True}()$. Then, $\exists v_2 = \text{True}(). \exists Y = 0. \text{isOVar} \ \#z \mapsto^+ \text{True}() \gg^0 \text{True}() \wedge Y < 1$.

18. For the workable $\text{isOVar } v$, where $v \neq \#z$, $\text{isOVar } v \ggg^1 \text{False}()$. Then, $\exists v_2 = \text{False}(). \exists Y = 0. \text{isOVar } v \mapsto^+ \text{False}() \ggg^0 \text{False} \wedge Y < 1$.
19. For the workable $\text{isOVar } w$ it is easy to show that it could never be reduced to a value in a single parallel-reduction step, so the property vacuously holds.

□[e.p.]

7.5 Permutation Lemma

Lemma 38 (Permutation) $\forall X \in \mathbb{N}. \forall w_1, w_2 \in W, e_1 \in \mathbb{E}$.

$$w_1 \ggg^X w_2 \mapsto e_1 \implies (\exists e_2 \in \mathbb{E}. w_1 \mapsto^+ e_2 \ggg e_1).$$

Proof (Lemma 38). $\forall X \in \mathbb{N}. \forall w_1, w_2 \in W, e_1 \in \mathbb{E}$.

$$w_1 \ggg^X w_2 \mapsto e_1 \implies (\exists e_2 \in \mathbb{E}. w_1 \mapsto^+ e_2 \ggg e_1).$$

Proof is by induction on the complexity X of derivation $w_1 \ggg^X w_2$, and then my the size of w_1 . Proof proceeds by case analysis over derivation of $w_1 \ggg^X w_2$.

1. For the workable $(\lambda x.e) v$, $\frac{e \ggg^M e' \quad v \ggg^N v'}{(\lambda x.e) v \ggg^{M+\#(x,e')N+1} e'[x:=v']}$, and $e'[x:=v] \mapsto e_1$.

By Lemma 18, $e[x:=v] \ggg^{M+\#(x,e')N} e'[x:=v]$. Since $M + \#(x,e')N < M + \#(x,e')N + 1$, and by monotonicity properties $e'[x:=v]$ is a workable, then the induction hypothesis can be applied to $e[x:=v]$ to obtain: $\exists e_2. e[x:=v] \mapsto^+ e_2 \ggg e_1$. Then, it easily follows: $\exists e_3 = e_2. (\lambda x.e) v \mapsto^+ e[x:=v] \mapsto^+ e_3 \ggg e_1$.

2. For the workable $(\lambda(\#z \Rightarrow x).e) (\#z \Rightarrow b)$, $\frac{e \ggg^M e'}{(\lambda(\#z \Rightarrow x).e) (\#z \Rightarrow b) \ggg^{M+1} e'[x:=\lambda x'.b[\#z:=x']]} \mapsto e_1$.

By Lemma 18, $\exists Z. e[x:=\lambda x'.b[\#z:=x']] \ggg^Z e'[x:=\lambda x'.b[\#z:=x']] \wedge Z \leq M$. Then, by the induction hypothesis: $\exists e_2. e[x:=\lambda x'.b[\#z:=x']] \mapsto^+ e_2 \ggg e_1$. Then, it immediately follows $\exists e_3 = e_2. (\lambda(\#z \Rightarrow x).e) (\#z \Rightarrow b) \mapsto^+ e[x:=\lambda x'.b[\#z:=x']] \mapsto^+ e_3 \ggg e_1$.

3. For the workable $w_1 e \ggg^{M+N} w'_1 e' \mapsto e_1$, $\frac{w_1 \ggg^M w'_1 \quad e \ggg^N e'}{w_1 e \ggg^{M+N} w'_1 e'}$.

There are five possibilities, and they all must be examined.

- (a) $w_1 e_1 \ggg^X w'_1 e'_1 \mapsto e_2 e'_1$ By definition of \ggg , $w_1 \ggg^M w'_1$, and $w'_1 \mapsto e_2$ Then, by the induction hypothesis: $\exists e_3. w_1 \mapsto^+ e_3 \ggg e_2$. Then, $\exists e_4 = e_3 e_1. w_1 e_1 \mapsto^+ e_4 \ggg e_2 e'_1$.
- (b) $w e \ggg^X v w' \mapsto v w''$ By definition of \ggg , $e \ggg^M w'$, and by definition of \mapsto , also $w' \mapsto w''$. Then, by monotonicity of \ggg , e must be a workable, and the induction hypothesis can be applied to it: $\exists e_3. e \mapsto^+ e_3 \ggg w''$. Since $w \ggg v$, then by transition lemma: $\exists v_2. w \mapsto^+ v_2 \ggg v$. Then, $\exists e_4 = v_2 e_3. w e \mapsto^+ v_2 e \mapsto^+ v_2 e_3 \ggg v w''$.
- (c) $w e \ggg^X (\lambda x.e_1) v \mapsto e_1[x:=v]$ There are two possibilities:
- $e \in \mathbb{V}$. Then, applying transition to w gives: $\exists v_2 = \lambda x.e_4. w \mapsto^+ (\lambda x.e_4) \ggg (\lambda x.e_1)$. Then, $\exists e_3 = (\lambda x.e_4) e. w e \mapsto^+ (\lambda x.e_4) e \ggg e_1[x:=v]$.
 - $e \in \mathbb{W}$. Then, transition can be applied to both w and e , to obtain $\exists v_1 = \lambda x.e_4. w \mapsto^+ \lambda x.e_4 \ggg \lambda x.e_1$ and $\exists v_2. e \mapsto^+ v_2 \ggg v$. Then $\exists e_5 = v_1 v_2. w e \mapsto^+ v_1 e \mapsto^+ v_1 v_2 \ggg e[x:=v]$.
- (d) $w e \ggg^X (\lambda(\#z \Rightarrow x).e_1) (\#z \Rightarrow b) \mapsto e_1[x:=\lambda x'.b[\#z:=x']]$ Similar to previous case.
- (e) $w e \ggg^X (\lambda^f \in F \cup \{k\} f x_f.e_f) (k v) \mapsto e[x:=v]$ Similar to previous case.
4. For the workable $v w$, there are four cases:
- (a) $v w \ggg^{M+N} v' w' \mapsto v' w''$ Then, by definition of \ggg , $v \ggg^M v'$ and $w \ggg^N w'$. By the induction hypothesis we have: $\exists e_2. v \mapsto^+ e_2 \ggg w''$. Then, $\exists e_3 = v e_2. v w \mapsto^+ v e_2 \ggg v' w''$.

- (b) $(\lambda x.e_1) w \xrightarrow{M+N} (\lambda x.e'_1) v \mapsto e'_1[x:=v]$ Since $w \gg v$, we can apply transition lemma. Thus, $\exists v_2.w \mapsto^+ v_2 \gg v$.
Now, $\exists e_2 = (\lambda x.e_1) v_2. (\lambda x.e_1) w \mapsto^+ (\lambda x.e_1) v_2 \mapsto^1 e_1[x:=v_2]$. Then, by the substitution lemma: $e_1[x:=v_2] \gg e'_1[x:=v]$, since $e_1 \gg e'_1$ and $v_2 \gg v$.
- (c) $(\lambda(\#z \Rightarrow x).e_1) (\#z \Rightarrow w) \xrightarrow{M+N} (\lambda(\#z \Rightarrow x).e'_1) (\#z \Rightarrow b) \mapsto e'_1[x:=\lambda x.b[\#z:=x']]$ Similar to previous case.
- (d) $(\lambda^{f \in F \cup \{k\}} f x_f.e_f) (k w) \gg (\lambda^{f \in F \cup \{k\}} f x_f.e'_f) (k v) \mapsto e'_k[x:=v]$ Similar to previous case.
5. $f w \xrightarrow{X} f w' \mapsto f e$ Then, by the definition of \gg , $w \xrightarrow{X} w' \mapsto e$.
By the induction hypothesis, $\exists e_1. w \mapsto^+ e_1 \gg e$. Then, $\exists e_2 = f e_1. f w \mapsto^+ f e_1 \gg f e$.
6. $\#z \Rightarrow w \xrightarrow{X} \#z \Rightarrow w' \mapsto \#z \Rightarrow e$. Then, by definition of \gg , $w \gg w' \mapsto^+ e$. By the induction hypothesis $\exists e_1. w \mapsto^+ e_1 \gg e$. Then, $\exists e_2 = \#z \Rightarrow e_1. \#z \Rightarrow w \mapsto^+ \#z \Rightarrow e_1 \gg \#z \Rightarrow e$.
7. For the workable (w, e) there are two possibilities:
- (a) $(w, e) \xrightarrow{M+N} (w', e') \mapsto (w'', e')$ Then, by the induction hypothesis: $\exists e_1. w \mapsto^+ e_1 \gg w''$.
Then, $\exists e_2 = (e_1, e). (w, e) \mapsto^+ (e_1, e) \gg (w'', e')$.
- (b) $(w, e) \xrightarrow{M+N} (v, w') \mapsto (v, w'')$. Then, e must be a workable, and $e \gg w' \mapsto w''$. Thus, by the induction hypothesis $\exists e_1. e \mapsto^+ e_1 \gg w''$. Furthermore, by the transition lemma: $\exists v_1. w \mapsto^+ v_1 \gg v$.
Then $\exists e_2 = (e_1, v_1). (w, e) \mapsto^+ (v_1, e) \mapsto^+ (v_1, e_1) \gg (v, w'')$.
8. For the workable (v, w) , where $(v, w) \gg (v', w') \mapsto (v', w'')$. By the induction hypothesis $\exists e_1. w \mapsto^+ e_1 \gg w''$.
Then $\exists e_2 = (v, e_1). (v, w) \mapsto^+ (v, e_1) \gg (v', w'')$.
9. For the workable $w = e$, there are four possibilities:
- (a) $w = e \gg w' = e' \mapsto w'' = e'$ Then $w \gg w' \mapsto w''$. By the induction hypothesis: $\exists e_1. w \mapsto^+ e_1 \gg w''$.
Then $\exists e_2 = (e_1 = e). w = e \mapsto^+ e_1 = e \gg w'' = e'$.
- (b) $w = e \gg \#z = w' \mapsto \#z = e'$ By transition lemma $\exists v_1. w \mapsto^+ v_1 \gg \#z$. Since $e \gg w'$, by transitivity properties of \gg , e must also be a workable, and, as $w' \mapsto e'$, the induction hypothesis applies: $\exists e_1. e \mapsto^+ e_1 \gg e'$. Then $\exists e_2 = (v_1 = e_1). w = e \mapsto^+ v_1 = e \mapsto^+ v_1 = e_1 \gg \#z = e'$.
- (c) $w = e \gg \#z = \#z \mapsto \text{True}()$ By transition lemma and properties of parallel reduction, $w \mapsto^+ \#z$ and $e \mapsto^+ \#z$. Then $w = e \mapsto^+ \#z = e \mapsto^+ \#z = \#z \gg \text{True}()$.
- (d) $w = e \gg \#z = \#z' \mapsto \text{False}()$ By transition lemma and properties of parallel reduction, $w \mapsto^+ \#z$ and $e \mapsto^+ \#z'$. Then $w = e \mapsto^+ \#z = e \mapsto^+ \#z = \#z' \gg \text{False}()$.
10. For the workable $\#z = w$, there are three possibilities:
- (a) $\#z = w \mapsto \#z = w' \mapsto \#z = w''$. By the induction hypothesis, $\exists e_1. w \mapsto^+ e_1 \gg w''$. Then $\exists e_2 = (\#z = e_1). \#z = w \mapsto^+ \#z = e_1 \gg \#z = w''$.
- (b) $\#z = w \mapsto \#z = \#z \mapsto \text{True}()$ By the transition lemma $\exists v. w \mapsto^+ v \gg \#z$. But v must be $\#z$, if it parallel reduces to $\#z$. So, $\exists e_2 = (\#z = \#z). \#z = w \mapsto^+ \#z = \#z \xrightarrow{1} \text{True}()$.
- (c) $\#z = w \mapsto \#z = \#z' \mapsto \text{False}()$ By the transition lemma $\exists v. w \mapsto^+ v \gg \#z'$. But v must be $\#z'$, if it parallel reduces to $\#z'$. So, $\exists e_2 = (\#z = \#z'). \#z = w \mapsto^+ \#z = \#z' \xrightarrow{1} \text{False}()$.
11. For the workable $\#z = \#z'$, there are two possibilities:
- (a) $\#z = \#z' \xrightarrow{0} \#z = \#z' \mapsto \text{True}()$. Obviously $\exists e_1 = \text{True}(). \#z = \#z' \mapsto^+ \text{True}() \gg \text{True}()$.
- (b) $\#z = \#z' \xrightarrow{0} \#z = \#z' \mapsto \text{False}()$. Obviously $\exists e_1 = \text{False}(). \#z = \#z' \mapsto^+ \text{False}() \gg \text{False}()$.
12. For the workable, $\pi_1 w$ there are two possibilities:
- (a) $\pi_1 w \gg \pi_1 w' \mapsto \pi_1 w''$. Then, by the induction hypothesis, $\exists e_1. w \mapsto^+ e_1 \gg w''$. From this it easily follows $\exists e_2 = \pi_1 e_1. \pi_1 w \mapsto^+ \pi_1 e_1 \gg \pi_1 w''$.
- (b) $\pi_1 w \gg \pi_1 (v_1, v_2) \mapsto^+ v_1$. Then, by transition lemma, $\exists v_3. w \mapsto^+ (v_3, v_4) \gg (v_1, v_2)$. Then, $\exists e_2 = \pi_1(v_3, v_4). \pi_1 w \mapsto^+ \pi_1(v_3, v_4) \gg v_1$.
13. The case for π_2 is symmetrical to the case above.
14. For the workable $\pi_1(v_1, v_2) \gg \pi_2(v'_1, v'_2) \mapsto v'_1$. Then $v_1 \gg v'_1$.
 $\exists e_2 = v_1. \pi_1(v_1, v_2) \mapsto^1 v_1 \gg v'_1$.
15. For the workable $\pi_2(v_1, v_2) \gg \pi_2(v'_1, v'_2) \mapsto v'_2$. Then $v_2 \gg v'_2$.
 $\exists e_2 = v_2. \pi_2(v_1, v_2) \mapsto^1 v_2 \gg v'_2$.
16. For the workable $\text{isOVar } \#z$, $\text{isOVar } \#z \xrightarrow{0} \text{isOVar } \#z \mapsto \text{True}()$. Then $\exists e_2 = \text{True}(). \text{isOVar } \#z \mapsto^+ e_2 \xrightarrow{0} \text{True}()$.

17. For the workable $\text{isOVar } v$, where $v \neq \#z$, $\text{isOVar } v \gg^X \text{isOVar } v' \mapsto \text{False}()$. Then $\exists e_2 = \text{False}(). \text{isOVar } v \mapsto^+ e_2 \gg^0 \text{False}()$.
18. For the workable $\text{isOVar } w$, there are three possibilities:
 - (a) $\text{isOVar } w \gg \text{isOVar } \#z \mapsto \text{True}()$. By definition of \gg , $w \gg \#z$. By transition $w \mapsto^+ \#z$. Then $\exists e_2 = \text{True}. \text{isOVar } w \mapsto^+ \text{isOVar } \#z \mapsto e_2 \gg \text{True}()$.
 - (b) $\text{isOVar } w \gg \text{isOVar } v \mapsto \text{False}()$, where $v \neq \#z$. By definition of \gg , $w \gg v$. By transition $\exists v'. w \mapsto^+ v' \gg v$. Then $\exists e_2 = \text{False}(). \text{isOVar } w \mapsto^+ \text{isOVar } v' \mapsto e_2 \gg \text{False}()$.
 - (c) $\text{isOVar } w \gg \text{isOVar } w' \mapsto \text{isOVar } w''$. By definitions of \gg and \mapsto , $w \gg w' \mapsto w''$. By the induction hypothesis, $\exists e. w \mapsto^+ e \gg w''$. Then, $\exists e_2 = \text{isOVar } e. \text{isOVar } w \mapsto^+ \text{isOVar } e \gg \text{isOVar } w''$.

□[e.p.]

References

1. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.
2. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000, USA. Available online from <ftp://cse.ogi.edu/pub/tech-reports/README.html>. Last viewed August 1999.
3. G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
4. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Revised October 99. Available from author (taha@cs.chalmers.se).
5. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *2000 SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, January 2000.
6. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997. An extended and revised version appears in [8].
7. Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. Technical Report CSE-99-007, Department of Computer Science, Oregon Graduate Institute, January 1999. Extended version of [6]. Available from [2].
8. Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000. In Press. Revised version of [7].
9. Masako Takahashi. Parallel reductions in λ -calculus. *Information and Computation*, 118(1):120–127, April 1995.

8 Acknowledgements

The third author would like to thank David Sands and Thierry Coquand for useful discussions that have influenced this work positively.

APPENDIX D

Constructing Modular Type Systems

Chiyan Chen

Advisor: Mark P. Jones

Computer Science and Engineering Department
Oregon Graduate Institute of Science and Technology

July 26, 2000

Abstract

The use of domain specific languages (DSLs) and advanced programming language type systems can have a significant effect on programmer productivity and software quality. However, the high cost of designing a type system and the limited user community of a DSL makes it economically infeasible for us to design typed DSLs and take the advantage of both. In this paper, we address this problem by reducing the effort that is needed to build type systems.

Most current type systems are designed in a monolithic fashion—with several language features mixed up together in a single block—which can make them hard to understand, hard to reuse, and hard to extend. In this paper, we show a way of building type systems in a modular fashion by composing independent and reusable building blocks, each of which implements one particular type feature. Using this way, building type systems becomes as simple as properly selecting and composing building blocks.

The work presented here is, to our knowledge, the first attempt to build type systems in a compositional style. Technical contributions of this work include a novel use of rank-2 polymorphism, and the introduction of a new form of map operator over abstract syntax.

1 Introduction

One of the goals of programming language research is to design languages that will enable programmers to develop code more efficiently, while also producing more reliable software systems. Domain specific languages (DSLs) and programming language type systems are two distinct threads of current research that potentially realize this goal. The strengths of DSLs are in allowing programmers to work at a higher level, using the familiar abstractions and notations with which they are already familiar instead of the constructs of a general purpose language. The strengths of programming language type systems are in helping to guarantee type-safe execution, to justify code optimizations, and to document programmers' intentions. Each of these can have a significant effect on programmer productivity and on the quality of the final product. It follows that typed DSLs, offering the benefits of both in a single language, would make particularly attractive tools for a wide range of programming tasks. However, DSLs usually only have a limited range of users, while building type systems usually require a lot of effort. So the high starting cost makes it economically infeasible to build a type system for each DSL.

In this paper we propose a possible solution to address the problem by trying to reduce the efforts to build type systems. Traditionally, type systems are built in a monolithic fashion, with all the type features mix together in a single block. This way of building type system makes them hard to reuse and hard to extend. We propose an innovative approach to build reusable type systems in a modular fashion. The key idea is as

follows: we design some independent and reusable type system building blocks, each of which has a standard interface for extension, and each of which implements one particular type feature, such as primitive types, λ -calculus[1], polymorphism[2], etc; when we want to build a type system, we first select the type system building blocks that implement the type features we desire to have, and then compose them by their standard extension interface to obtain the type system as needed. By this means, it becomes more easier to build a type system, and makes it economically feasible to build typed DSLs.

The approach of building modular type systems also brings other advantages besides the principle motivation of offering an economical way of building typed DSLs.

- **Modular type systems are easier to understand.** It is often easier to understand several small simple blocks than to understand a big complicated one.
- **Modular type systems enhance reusability.** The building blocks of a modular type system are designed with a standard interface for extension, so they can be easily reused in other settings. The type features in a monolithic type system are tightly coupled together, so it is hard to abstract them out of the framework and make them reused.
- **Modular type systems are easier to extend.** To extend a modular type system, we just need to design a new building block for the newly added type feature, and compose it with the original type system using its standard extension interface; however, to extend a monolithic type system, we usually need to modify the overall system, and carefully fit the new type feature in such that it interacts correctly with the existing framework.
- **The framework of modular type systems can help to study the interaction between type features.** The interaction between type features are reflected in the different results obtained by composing the same set of building blocks in different orders.

In Section 2, we will show how to build a modular type system in a compositional style for a simple toy DSL; in Section 3 and Section 4, we will give a modular implementation for the type system described in Section 2, where Section 3 deal with syntactic problem and Section 4 deal with static semantics problem; in Section 5, we will discuss some related works of this paper; finally in Section 6, we draw a conclusion and sketch some future work.

2 Type Systems in a Compositional Style

We consider a programming language type system to be a combination of two components: syntax and static semantics. Syntax consists of expression syntax and type syntax of the programming language, and static semantics is a typing relationship between these two kinds of syntaxes. An expression is said to be well-typed if there exists a type associated with it in the static semantics; otherwise it is ill-typed. Normally the static semantics are delineated by a set of axioms and inference rules. An expression is said to be of a certain type if there exists a typing derivation using the typing rules asserting that. In this paper, we only consider the case that the typing relationship is functional from the expression syntax to the type syntax, so we will focus on type checking functions for static semantics.

In this section we will show an example of building a modular type system for a simple toy language in a compositional style. This will be done in three steps, each of which is associated with a particular type feature. Then we will extract a pattern of modularity from the example.

2.1 A Type System for the Language of Integer Arithmetic

We will first introduce the type system for a very simple language: integer arithmetic. In this language, there are only two kinds of expressions: integer constants and addition operations, and one type: *Int*. The expression syntax and type syntax are defined by context free grammar as follows:

$$\begin{array}{lll}
 \text{expression syntax : } E & = & \text{Integer}E \\
 & \text{Integer}E & = n \\
 & & | E + E \\
 \\
 \text{type syntax : } T & = & \text{Integer}T \\
 & \text{Integer}T & = \text{Int}
 \end{array}$$

In these definitions, *E* stands for the expression syntax, while *T* stands for type syntax. Note that currently, *E* only includes *IntegerE*, which stands for the expression syntax for integer arithmetic; *T* only includes *IntegerT*, which stands for the type syntax for integer arithmetic. In later sections, we will see that both *E* and *T* will get extended with more syntax structures. In the definition of *E*, *n* stands for integer constants. Having shown the syntax, we show the inference rules for type checking the expression syntax constructs.

$$\frac{}{\Gamma \vdash n : \text{Int}} \quad (\text{integer constant})$$

$$\frac{\Gamma \vdash E_1 : \text{Int} \quad \Gamma \vdash E_2 : \text{Int}}{\Gamma \vdash E_1 + E_2 : \text{Int}} \quad (\text{addition operation})$$

The *integer constant* rule means that any integer constant is of type *Int*; the *addition operation* rule means that an addition expression is of type *Int* if both of its two operands are of type *Int*. In this type system, we can have typing derivations like the following:

$$\frac{\frac{}{\vdash 1 : \text{Int}} \quad \frac{}{\vdash 2 : \text{Int}}}{\vdash 1 + 2 : \text{Int}}$$

Note that there are no ill-typed expressions in this type system.

2.2 Adding Functions to the Language

The type system we have just defined is for such a simple language that it has little practical use. In this section we make the language more interesting by extending the simple language of integer arithmetic with functions. Like in most functional languages, we adopt the use the λ -calculus to express functions. So we extend the expression syntax with the λ -calculus expression constructs: expression variables, λ -abstractions and function applications; we also extend the type syntax with an function arrow construct. The expression syntax and type syntax of the new language is defined by context free grammar as follows:

$$\begin{array}{lll}
\text{expression syntax : } E & = & \text{Integer } E \\
& | & \text{Lambda } E \\
\text{Lambda } E & = & x \\
& | & \lambda x : T. E \\
& | & E \ E
\end{array}$$

$$\begin{array}{lll}
\text{type syntax : } T & = & \text{Integer } T \\
& | & \text{Lambda } T \\
\text{Lambda } T & = & T \rightarrow T
\end{array}$$

In these definitions, E and T still stand for expression syntax and type syntax respectively. E includes two kinds of expression syntaxes, $\text{Integer } E$ and $\text{Lambda } E$, while T includes two kinds of type syntaxes $\text{Integer } T$ and $\text{Lambda } T$. Both of them are extended with λ -calculus constructs on the base of integer arithmetic constructs. Then we show the inference rules for type checking the λ -calculus expression syntax constructs.

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \quad (\text{var})$$

$$\frac{\Gamma_x, x : T' \vdash E : T}{\Gamma \vdash (\lambda x : T'. E) : T' \rightarrow T} \quad (\lambda - \text{abstraction})$$

$$\frac{\Gamma \vdash E_1 : T' \rightarrow T \quad \Gamma \vdash E_2 : T'}{\Gamma \vdash E_1 \ E_2 : T} \quad (\text{function application})$$

Note that a type assignment context Γ , which maps expression variable names to types, is introduced into the type checking rules by the extension of λ -calculus. The *var* rule means that an expression variable is of the type that is assigned to it by the type assignment context; the $\lambda - \text{abstraction}$ rule means that $\lambda x : T'. E$ is of type $T' \rightarrow T$ in the context Γ if E is typed T in the context $\Gamma_x, x : T'$, where $\Gamma_x, x : T'$ specifies a type assignment context that maps all the expression variables to the same types as Γ but the variable x to the type T' ; the *function application* rule means that the function application expression $(E_1 \ E_2)$ is of type T in the context Γ , if E_1 is of type $T' \rightarrow T$ in the same context Γ , and E_2 is typed T' in the same context Γ .

The extension of functions to the original type system is static semantics preserving, in the sense that well-typed expressions in the original type system remain well-typed and ill-typed expressions in the original type systems remain ill-typed. Because there are new expression syntax constructs in the extended type system, there are more well-typed expressions, such as the following:

$$\frac{\frac{\frac{x : \text{Int} \vdash x : \text{Int} \quad x : \text{Int} \vdash 1 : \text{Int}}{x : \text{Int} \vdash x + 1 : \text{Int}}}{\vdash (\lambda x : \text{Int}. x + 1) : \text{Int} \rightarrow \text{Int}} \quad \overline{\vdash 2 : \text{Int}}}{\vdash (\lambda x : \text{Int}. x + 1) \ 2 : \text{Int}}$$

Also, there are more ill-typed expressions in the new type system, such as $x + 1$ (unbounded variable), $(2\ 1)$ (invalid application) and $\lambda x : Int \rightarrow Int. x + 1$ (type mismatch).

2.3 Adding Polymorphism to the Language

So far we have two type features in our type system: integer arithmetic and functions. In this section we make the use of functions more powerful by allowing polymorphism. The expression syntax is extended with new constructs for polymorphism: type abstractions and type applications; the type syntax is also extended with polymorphism constructs: type variables and polymorphic type schemes. The syntax of the this new language is given by context free grammars as follows:

$$\begin{array}{lll}
 \text{expression syntax : } E & = & IntegerE \\
 & | & LambdaE \\
 & | & PolyE \\
 PolyE & = & \Lambda\alpha.E \\
 & | & E\ T \\
 \\
 \text{type syntax : } T & = & IntegerT \\
 & | & LambdaT \\
 & | & PolyT \\
 PolyT & = & \alpha \\
 & | & \forall\alpha.T
 \end{array}$$

Now there are three kinds of syntaxes in both E and T . E is extended with $PolyE$ on the base of $IntegerE$ and $LambdaE$, while T is extended with $PolyT$ on the base of $IntegerT$ and $LambdaT$. The inference rules for type checking the expression syntax constructs for polymorphism are shown as follows:

$$\frac{\Gamma \vdash E : T \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda\alpha.E : \forall\alpha.T} \quad (\text{type abstraction})$$

$$\frac{\Gamma \vdash E : \forall\alpha.T}{\Gamma \vdash E\ T' : [T'/\alpha]T} \quad (\text{type application})$$

The *type abstraction* rule means that $\Lambda\alpha.E$ is of type $\forall\alpha.T$ in context Γ if E is of type T in the same context Γ and the type variable name α does not appear freely in Γ ; the *type application* rule means that $E\ T'$ is of type $[T'/\alpha]T$ in context Γ if E is of type $\forall\alpha.T$ in the same context Γ , where $[T'/\alpha]T$ is the type obtained by substituting all the free occurrence of the type variable α in the type T with the type T' . Like the extension of λ -calculus, the extension of polymorphism is also static semantic preserving. There are more well-typed expressions related with the polymorphism expression syntax constructs, such as the following:

$$\frac{\frac{x : \alpha \vdash x : \alpha}{\vdash \lambda x : \alpha. x : \alpha \rightarrow \alpha}}{\vdash \Lambda\alpha. \lambda x : \alpha. x : \forall\alpha. \alpha \rightarrow \alpha}$$

Also there are more ill-typed expressions, such as $\Lambda\alpha. \lambda x : \alpha. x + 1$ (type mismatch), $3\ Int$ (bad type application) and $(\Lambda\alpha. \lambda x : \alpha. x)\ 1$ (instantiation needed).

2.4 A Modular Pattern

In the previous three sections, we define a type system consisting of three type features: integer arithmetic, λ -calculus and polymorphism. This type system is developed in a compositional style by three steps. In the first step, we define the syntax for a simple language of integer arithmetic, and describe its static semantics by a set of type checking rules for its expressions; then in the second step, we extend the type system by adding λ -calculus syntax constructs, together with the static semantics for the newly added expressions; finally in the last step, we extend the type system again with new syntax constructs for polymorphism, and the corresponding static semantics for the new expressions. If we consider the first step as extending an empty type system (a type system with no syntax at all and thus no static semantics) with integer arithmetic syntax and static semantics, then all of the three step can be seen as some kind of extension to an existing type system with new syntax structures and new static semantics. We name these extension parts “building blocks”, for they are the components that build up the type system. By composing a building block with an existing type system, we will get a new type system extended with the type feature associated with the building block. In this sense, building blocks can also be seen as “type system transformers” that transform a type system to another. With the notion of building blocks, we can review the construction of the type system we defined from another perspective: starting from an empty type system, we put in three building blocks one by one for the type features of integer arithmetic, λ -calculus and polymorphism respectively. The idea can be captured using the following equation:

```
my_ts = mkTS (integerBlock . lambdaBlock . polyBlock)
```

where `my_ts` stands for the type system we built; “.” stands for some composition mechanism between building blocks; `mkTS` is a function that transforms the composition of building blocks to a type system. Later in Section 4, we will give the detailed implementation of “.” and `mkTS`.

This modular approach makes the construction of type systems very flexible. For example, if we observe that the type features of λ -calculus and polymorphism can be combined together to form System F[1], in which integer type and values can be encoded, we may decide not to have the type feature of integer arithmetic in our type system. This can be done by simply taking away the building block for integer arithmetic from the construction of the type system:

```
my_ts = mkTS (lambdaBlock . polyBlock)
```

For another example, if we only need a language that has the feature of integer arithmetic, logical operations and functions, we can design a new building block “`boolBlock`” for logical operations, and compose it with the building blocks for integer arithmetic and λ -calculus to form the type system we need:

```
my_ts = mkTS (integerBlock . boolBlock . lambdaBlock)
```

2.5 Summary

From the discussion we have made, we understand that a big and complex type system with many type features can be built by composing some independent and reusable building blocks, each of which implements a particular type feature. Different type systems can be built from different set of building blocks with different composition orders. This makes it easier to build type systems for DSLs. We can design a large set

of building blocks in advance, which cover the type features that might be needed. When we need to build a type system, we just select and compose the building blocks that implement its type features. In case we need a novel type feature that is not implemented by any available building block, we can design a new building block for it and then make use of it. The cost of designing a new building block might be high. However, because the building blocks are reusable, we do it once and for all.

3 Build Syntax for Modular Type Systems

In the following two sections, we will give an implementation in Haskell for the modular type system introduced in the previous section. As stated, a type system is a combination of syntax and static semantics. So as to build a modular type system, we need to build both its syntax and static semantics in a modular fashion. These two problems will be addressed separately. In this section we show how to deal with syntax. Syntax includes expression syntax and type syntax. We will show how to construct the type syntax in an evolutionary manner, and then give the construction of the expression syntax directly using the same idea.

3.1 Type Syntax

We want to build the syntax for a type system that consists of three type features: integer arithmetic, λ -calculus and polymorphism. A monolithic representation of type syntax is shown as follows:

```
datatype = INT          -- constant type INT
          | Fun Type Type -- function arrow
          | TVar String   -- type variable
          | Forall String Type -- polymorphic type
```

If we want to represent the type syntax in a modular fashion, we need to break this monolithic definition into three pieces for the three type features respectively. First we show a method of decomposing the syntax directly:

```
type Type    = IntType <+> LType <+> FType
data IntType = INT          -- type syntax for integer arithmetic
data LType   = Fun Type Type -- type syntax for lambda calculus
data FType   = TVar String   -- type syntax for polymorphism
              | Forall String Type

data a <+> b = L a | R b      -- a sum type constructor
```

Here we literally break the monolithic definition of `Type` into three pieces, each of which captures the type syntax of one type feature. These three pieces are composed with a `<+>` operator to form a new definition of `Type` that is essentially equivalent to the original monolithic one. However, there is a problem in this method: the definitions of `LType` and `FType` are mutually recursive with the definition of `Type`, which makes them depend on each other; as a result, it is hard for these definitions to appear independently in separate modules.

To address this problem, we refine our decomposition by parameterization. We use type parameters to abstract away the dependence of `LType` and `FType` on `Type`, and thus we reduce the coupling between these definitions.

```

type Type          = (IntTypeGen Type) <+> (LTypeGen Type) <+> (FTypeGen Type)
data IntTypeGen t = INT          -- open type syntax for integer arithmetic
data LTypeGen t   = Fun t t      -- open type syntax for lambda calculus
data FTypeGen t   = TVar String  -- open type syntax for polymorphism
                  | Forall String t

```

We will call these parameterized constructors, such as `IntTypeGen`, `LTypeGen` and `FTypeGen`, “open syntax”, in the sense that they do not prematurely determine what the final type syntax should be, but leave the type parameter as an interface for extension. In this new method of decomposition, the definition of the three open syntax pieces do not involve recursion at all, so they can exist independently; the definition of `Type` becomes self recursive, which simplifies the definition structure. However, note that there are some redundancies in the definition of `Type`, in which we repeat the use of `Type` three times for recursion.

To eliminate the redundancy, we lift the `<+>` operator to do the sum on two open syntaxes:

```

data (newTypeGen <+> oldTypeGen) t
  = NewType (newTypeGen t)
  | OldType (oldTypeGen t)

```

Now we can use the new `<+>` to compose the open syntax pieces.

```

type FinalTypeGen = IntTypeGen <+> LTypeGen <+> FTypeGen
type Type         = FinalTypeGen Type

```

By this means, we eliminate the redundancy problem. Furthermore, we observe that `Type` is actually the fix point of the constructor `FinalTypeGen`. We capture this point and refine the way of defining `Type` with a fix point operator.

```

data Tfix t = MkType (t (Tfix t))
type Type   = Tfix FinalTypeGen

```

The fix point operator `Tfix` shuts down the extension interface of an open syntax and turns it into a “closed syntax”. With this approach of defining type syntax, we gain the ability of writing functions on type syntax structures with more flexible types. For example, consider a function to test the syntactic equality of two pieces of type syntax. Without the `Tfix` operator, we can only give the following form of type to it,

```
Type -> Type -> Bool.
```

Because `Type` is defined in an ad-hoc way, this function type only works for one kind of type syntax composed by a particular set of open syntax pieces. However, with the help of `Tfix` operator, we can have a more generic type:

```
Tfix t -> Tfix t -> Bool.
```

Note that the type variable t is polymorphic in this type, so it can be instantiated to any composition of open syntaxes. As a result, the type of every function that test syntactic equality on type syntax can be an instance of this type.

3.2 Expression Syntax

We will use the same technique used to deal with type syntax to define expression syntax. First, we define three open syntax constructors.

```
data IntExprGen e t = IntLit Int      -- integer literals
                  | Add e e          -- addition expression
data LCEExprGen e t = Var String      -- expression variable
                  | Lam String t e    -- lambda abstraction
                  | App e e          -- function application
data FExprGen e t = TLam String e     -- explicit type abstraction
                  | TApp e t         -- type application
```

A difference between the expression open syntax and the type open syntax is that it is parameterized with two type variables, one for expression syntax, the other for type syntax. This is because the construction of some expression syntax requires the use of type syntax. For example, the expression syntax of λ -abstraction uses type syntax for the function parameter type.

The definition of the $\langle + \rangle$ operator of the expression syntax is just like that of the type syntax.

```
data (newExprGen <+> oldExprGen) e t
  = NewExpr (newExprGen e t)
  | OldExpr (oldExprGen e t)
```

The fix point operator of expression syntax is `Efix`, whose definition involves the `Tfix` operator. For a similar reason, `Efix` also takes two parameters, like the expression open syntax.

```
data Efix e t = MkExpr (e (Efix e t) (Tfix t))
```

The datatype `Expr` is defined as follows:

```
type FinalExprGen = IntExprGen <+> LCEExprGen <+> FExprGen
type Expr         = Efix FinalExprGen FinalTypeGen
```

4 Build Static Semantics for Modular Type System

Having shown how to build syntax, we will now show how to build the static semantics for the modular type system.

4.1 Type Checking Function

As stated, we consider the static semantics to be a type checking function. A type checking function should take an expression as argument and return a type corresponding to it. In most cases, simply return a type is not enough, because type checking is usually done in some contexts. For example, it is possible for ill-typed expressions to exist, so we need to maintain a context in which we can raise a bad type error during the process of type checking. For another example, we need to maintain a context of type assignment during the process of type checking λ -calculus expressions. So we must take the type checking contexts into account when describing the type of type checking functions. In this paper, we use monads[3] to capture the type checking contexts, so the type of type checking functions should be the following form:

`Expr -> CM Type,`

where CM denotes some context monad.

4.2 Open System, Closed System and Building Blocks

In order to illustrate our way of building modular static semantics, we need to introduce three notions first: open system, closed system and building block. An open system is an incomplete type checking function that has an standard interface for extension. Because it is incomplete, it does not supply any real type checking facilities. A building block is a function that takes an open system as argument and returns another open system extended with static semantics of a new type feature. It does the extension using the original open system's standard extension interface, and generate a new open system that exposes the same standard extension interface. Building blocks can also be thought of as open system transformers. So far, we get open systems and building blocks as open system transformers. By applying a series of building blocks to an open system, we can get an open systems. However, as known, open systems don't supply type checking facilities, so we need a final special function that shuts down the extension interface of an open system and generate a closed system that does supply the type checking facilities. We will call this special function `top`. Applying `top` to an open system, we will get a closed system which can not be extended any further but has the ability to do type checking work.

4.2.1 Data Representation of Closed Systems

We define a Haskell datatype for closed systems:

```
data Closed exprGen typeGen conGen
  = Closed {topTypeChk :: (Efix exprGen typeGen) ->
            conGen (Tfix typeGen) (Tfix typeGen)
            ... -- some auxiliary functions
            }
```

The datatype `Closed` is used to represent a closed system. It is parameterized with three type variables, `exprGen`, `typeGen` and `envGen`. The parameter `exprGen` is a constructor that specifies an expression open syntax, such as `(IntExpr <+> LCEExpr)`; the parameter `typeGen` is a constructor that specifies a type open

syntax, such as `(IntType <+> LType)`; the `envGen` parameter specifies a parameterized monad that captures the type checking context. It is parameterized by a type closed syntax because some type checking contexts may be related with type structure of the type system. For example, the type assignment environment context for λ -calculus can be considered as a finite function from a set of variable names to types. The `Closed` datatype is a record type that contains a `topTypeChk` function field. A closed system offers the type checking facility by its `topTypeChk` field. The `topTypeChk` function takes a expression closed syntax as argument and returns its type in a closed syntax form which is encapsulated in the type checking context monad.

Besides the `topTypeChk` function, we also have some auxiliary fields, which will be introduced later as they are needed.

4.2.2 Data Representation of Open Systems

Also, we define a Haskell datatype for open systems:

```
data Open exprGen typeGen conGen
  = Open {typeChk :: forall e t c. ErrorMonad (c (Tfix t)) =>
          Closed e t c ->
          SubType (typeGen (Tfix t)) (Tfix t) ->
          LiftM (conGen (Tfix t)) (c (Tfix t)) ->
          exprGen (Efix e t) (Tfix t) -> c (Tfix t) (Tfix t),
          ... -- some auxiliary functions
  }
```

The datatype `Open` is also parameterized with variables, which have the same name and meaning as those of the datatype `Closed`. The `Open` datatype is also a record type. Its contains a `typeChk` function field. A value whose type is of the form `Open exprGen typeGen envGen` is an open system that exposes an extension interface by its `typeChk` function field. The function `typeChk` is explicitly defined to be polymorphic. Its type is universally quantified by three type variables `e`, `t` and `a`. This kind of definition is known as rank-2 polymorphism[4]. The type variables `e` and `t` specify the top-level open syntaxes that are ready to be converted to closed syntaxes by fix point operators `Efix` and `Tfix`. The type variable `c` specifies the top-level parameterized context monad which will be use as the type checking context in the final closed system. These three type variables will be instantiated when an open system is transformed to a closed system.

An open system offers the standard extension interface by its `typeChk` function field. The function `typeChk` has a type class predicate and three arguments. It returns a value whose type is also a function type. In the following, we will explain the meaning of the types of the arguments and the return value, and show their use.

The type class predicate restricts the context monad to have the ability of raising an error during the process of type checking. Note that `c (Tfix t)` specifies the top level type checking context monad. The constructor class `ErrorMonad` is defined as follows:

```
class Monad m => ErrorMonad m where
  failE :: m a
  liftE :: Maybe a -> m a
```

The overloading function `failE` is to be used to raise an error; the overloading function `liftE` is to be used to lift a `Maybe` monad value to the an `ErrorMonad` value.

The parameter of the type `Closed e t c` represents a top-level closed system that will be built on the base of the current level of open system. It is passed back from the top level down to the current level because we need the top level type checking facility to construct the current level `typeChk` function. The use of rank-2 polymorphism gives us the ability of using a closed system without prematurely determine the actual value of its three type parameters `e`, `t` and `c`. These values will be determined when the open system has developed full-fledged and is transformed to a closed system.

The parameter of the type `SubType (typeGen (Tfix t)) (Tfix t)` specifies a explicit subtyping relationship between the current level of type open syntax and the top level type closed syntax. It is needed here because we do transformation between current level and top level type syntax in order to construct the `typeChk` function. The `SubType` relation is defined as follows:

```
data SubType a b
  = SubType {inj :: a -> b,
             proj :: b -> Maybe a}
```

The function `inj` injects a type open syntax of the current level into a top level type closed syntax; the function `proj` projects a top-level type closed syntax to a type open syntax of the current level.

The parameter of type `LiftM (envGen (Tfix t)) (a (Tfix t))` serves as a lift operator that lifts an operation defined in the current level context monad into the top level context monad. It is specially defined for lifting the operations of environment monads, because most type checking is done with in environment-like monads. The definition of the datatype `LiftM` is as follows:

```
data LiftM m n
  = LiftM {liftRdEnv :: forall a.m a -> n a,
          liftInEnv :: forall a b.(a -> m b -> m b) ->
                        (a -> n b -> n b)
  }
```

Normally, there are two non-standard operations[5] in an environment monad, one of which is to read out the current environment, the other is to enforce a computation to be done in a new environment. The function `liftRdEnv` lifts the operation of reading the current environment; the function `liftInEnv` lifts the operation of doing a computation in a new environment.

The return value of the `typeChk` function is of the type `exprGen (Efix e t) (Tfix t) -> a (Tfix t) (Tfix t)`. This is a function type which is much like the type of the `topTypeChk` function defined in the datatype `Closed`. The return value can be considered as the current level type checking function. It takes a value of the current level open expression syntax as argument and returns its type in a close syntax form which is encapsulated in the top level context monad.

Like closed systems, the datatype `Open` also has some corresponding auxiliary fields which will be introduced as needed.

4.2.3 Building Blocks, top and bot

Building blocks are open system transformers, whose type will be of the form:

`Open e t c -> Open e' t' c'.`

Because building blocks are functions, we can simply use function composition to compose two building blocks to obtain a larger composite building block. For example, if we have a building block of type:

`Open e t c -> Open e' t' c',`

and another building block of type:

`Open e' t' c' -> e'' t'' c'',`

then we can compose them using function composition to obtain a larger building block of type:

`Open e t c -> Open e'' t'' c''.`

Till now we know that the composition between building blocks is right function composition.

As indicated previously, we need to have a special function `top` whose type will be of the following form:

`Open e t c -> Closed e t c.`

The function `top` actually serves as a fix point operator, just like `Efix` and `Tfix`. It transforms an open system to a closed one by computing its fix point. In our framework of modular type systems, every closed system is a fixed point of some open system.

So far, we have only considered system transformers, besides which we also need an initial base open system to start with in order to finally construct a closed system. This base open system is called `bot` in our framework of modular type system.

With all the notions defined in this section, the mechanism to build the static semantics for the modular type system is clear. Given a set of building blocks, `b1`, `b2`, `b3`, we first use function composition to compose them to obtain a big composite building block; then we apply this big building block to the base open system `bot` to get an open system; finally we apply `top` to the open system to get its fix point as a closed system. The final closed system includes the static semantics for type features associated with the building blocks `b1`, `b2` and `b3`. This idea can be shown by the following expression:

`my_ts = top ((b1 . b2 . b3) bot).`

The value `my_ts` stands for a modular type system. The `(".")` operator used here is just the built-in Haskell function composition operator. Now we can define the `mkTS` function as follows:

`mkTS x = top (x bot)`

Then we can rewrite the definition of `my_ts` as:

```
my_ts = mkTS (b1 . b2 . b3)
```

4.3 Implementation of the Building Blocks

Now we give the implementation of the three building blocks associated with the type features of integer arithmetic, λ -calculus and polymorphism respectively.

4.3.1 A Building Block for Integer Arithmetic

We will start by showing the building block for integer arithmetic.

```
integerBlock :: Open e t c -> Open (IntExprGen <+> c) (IntTypeGen <+> t) c
integerBlock l
  = let typeChk_ = ...                -- Definition of type checking function.
    in Open {typeChk = typeChk_}
```

The `integerBlock` extends an open system with new syntax structures of integer arithmetic. The parameterized context monad `c` is not extended, because this building block does not add any new expression variable syntax into the expression syntax structure. The definition of the `typeChk` function can be divided into two parts, one for type checking the new expression syntax add by `integerBlock`, the other for type checking the old expression syntax inherited from the original open syntax. Here We show a sketch of the definition of the type checking function. This will be divided into two parts, one for type checking the new expression syntaxes added by the current building block, the other for type checking the old expression syntaxes in the original open system.

```
typeChk_ top s _ (NewExpr nt)
  = let te      = (topTypeEq top)      -- Closed type syntax equality comparison.
    topTC      = (topTypeChk top)      -- Top-level type checking.

    in case nt of
      IntLit _      -- Type check integer literals.
        -> return (((inj s).NewType) INT)

      Add t1 t2      -- Type check addition expression.
        -> do typ1 <- topTC t1          -- Type check first operand.
            typ2 <- topTC t2          -- Type check second operand.
            if (typ1 'te' (((inj s).NewType) INT)) && -- If both of the two operands
              (typ2 'te' (((inj s).NewType) INT))    -- are typed INT, then return
            then return (((inj s).NewType) INT)      -- INT, otherwise raise an
            else failE                               -- error.
```

In type checking the new expressions, we implement the two typing rules for integer arithmetic expressions, one for type checking integer constants, the other for type checking addition expressions. In this definition, **top** specifies the top level closed system, and **s** specifies the encoding of the subtyping relationship between the current level open type syntax and the top level closed type syntax. Two local functions, **te** and **topTC** are defined to help the implementation. The function **topTC** represents the **topTypeChk** field of the argument **top**, which specifies the top level closed type checking function. The function **te** represents the **topTypeEq** function field of the argument **top**, which tests the syntactic equality of two pieces of top level closed type syntax. Note that there is actually an auxiliary function field **typeEq** in the open systems and a corresponding function field **topTypeEq** in the closed systems. This auxiliary function is designed to test the syntactic equality of the type syntaxes. Its definition is omitted in this paper because of the little relevance to the essential idea and the straightforward implementation. The phrase **(inj s).NewType** injects the current level open type syntax (**INT** here) to the top level closed type syntax.

```

typeChk_ top s lf (OldExpr ot)
  = let subtype
      = let inj_ = (inj s).OldType
          -- Inject lower level open type
          -- syntax to top level closed type
          -- syntax.
          proj_ x
            = do x1 <- (proj s) x
              case x1 of
                OldType t -> return t
                -         -> failE
          -- Project top level closed type
          -- syntax to lower level open type
          -- syntax.
      in SubType {inj = inj_,
                  proj = proj_}
          -- Subtype relation between
          -- lower level and top level.

  in (typeChk 1) top subtype lf ot
          -- Call the lower level typeChk.

```

In type checking the old expressions, we make use of the function **integerBlock**'s argument **1**, which specifies the original open system. Essentially, we use its **typeChk** function field to type check the old expression syntax inherited from the original open system. To call the lower level **typeChk** function, we just pass the argument **top** directly. However, we need to pass newly constructed subtyping encoding argument and lifting encoding argument to fit the type of the lower level **typeChk** function. Here the new subtyping encoding is defined on the base of the current level subtyping encoding, while the lifting encoding stays unchanged because this building block does not extend the parameterized context monad at all.

From the definitions shown in this section, we can summarize a clear strategy to design the **typeChk** function field of open systems: implement the typing rules for type checking the new expression syntaxes added by the current building block, and use the lower level **typeChk** function field of the original open system to deal with the old expression syntaxes inherited from the original open system.

4.3.2 A Building Block for λ -Calculus

We continue to show a more complex building block, which is for λ -calculus.

```

data LCCon t = LCCon [(String, t)]    -- New environment type: each pair
                                       -- associates a type with a string
                                       -- that represents a variable name.

data EnvT r m t a = EnvT (r t -> m t a) -- Environment monad transformer.
applyEnvT (EnvT f) = f

lambdaBlock :: (MonadGen c) =>
  Open e t c -> Open (LCEExprGen <+> e) (LCTypeGen <+> t) (EnvT LCCon c)
lambdaBlock l
  = let typeChk_ = ...
    in Open {typeChk = typeChk_}

```

The `lambdaBlock` extends an open system with new syntax structures of λ -calculus. Unlike the `integerBlock`, it also extends the parameterized context monad with a layer of environment monad using the monad transformer `EnvT`. This is because a type assignment environment is needed to type checking the expressions in λ -calculus. A datatype `LCCon` specifies the type assignment environment added by `lambdaBlock`. It is defined as a list of pairs of strings and type closed structures, where the strings specify expression variable names. The type class predicate `MonadGen c` restricts the type parameter `c` of the original open syntax to be a parameterized monad. The detailed definition of the `EnvT` datatype and the `MonadGen` constructor class will be attached in the appendix.

The design of the `typeChk` function for `lambdaBlock` follows the basic strategy stated when constructing `integerBlock`. We divide the function into two parts. In type checking the new expressions syntaxes added by the current building block, we implement the typing rules for each of them; in type checking the old expression syntaxes inherited from the original open system, we make use of the lower level `typeChk` function. A sketch of the definition of the `typeChk` function is shown as follows.

```

typeChk_ top s lf (NewExpr nt) -- lf is the lift operator that lifts the two
                                -- nonstandard operations of current level context
                                -- monad up to the top level

= let te = (topTypeEq top)
    topTC   = (topTypeChk top)
    rdEnv    = EnvT (\(LCEEnv r) -> return r)    -- Current level rdEnv.
    inEnv r m = EnvT (\r1 -> applyEnvT m r)      -- Current level inEnv.
    lookup   = ...
    match    = ...
    update   = ...
  in case nt of
    Var x
      -> do env <- (liftRdEnv lf) rdEnv    -- Get current environment.
          lookup x env                    -- Look up a expression variable's type.
    App e1 e2
      -> do f <- topTC t1                  -- Type check the function.
          x <- topTC t2                    -- Type check the argument.
          r <- liftE (match f x)           -- Match the argument type
                                           -- against the function type.
      return r
    Lam v t e

```

```

-> do env <- (liftRdEnv lf) rdEnv      -- Get current environment.
    r <- ((liftInEnv lf) inEnv)      -- Type check the function
        (LCEnv (update (v, b) env)) -- body in an updated environment.
        (topTC t)
    return (((inj s).NewType) (Fun b r))

```

In type checking the new expressions, the meaning of the arguments `top` and `s` is as same as in the definition of `integerBlock`. The argument `lf` specifies the encoding of a lift operator which is used to lift the non-standard operations of the current level context monad into the top level context monad. The meaning of the local functions `te` and `topTypeChk` in this definition is as same as in the definition of `integerBlock`. The local functions `rdEnv` and `inEnv` are related with the type assignment context of the type checking. They are actually the two non-standard operations of the current level environment monad. The function `rdEnv` is used to read out the environment context at the point it is called; while the function `inEnv` is used to force a type checking to be done in a particular given type assignment context. The function `rdEnv` is used in type checking the expression variables and λ -abstractions. When it is used, it has to be lifted using the phrase `(liftRdEnv lf)` to fit the top level type checking context monad. The function `inEnv` is used in type checking λ -abstractions. When it is used, it has to be lifted using the phrase `(liftInEnv lf)` to fit the top level type checking context monad. There are also three other local functions: `lookup`, `match` and `update`. The function `lookup` is used to lookup the type of a given expression variable in the type assignment environment. The function `match` is used to match a given type against the argument part of a function arrow type, and returns the result part of the same function arrow type. The function `update` is used to update the type assignment environment with new bindings of expression variables and types. The implementations of these three functions are omitted here because they are quite straightforward.

```

typeChk_ top s lf (OldExpr ot)
  = let subtype = ...
      newlf
      = let liftRdEnv_ re = (liftRdEnv lf) (EnvT (\_ -> re))
          liftInEnv_ ine
            = (liftInEnv lf) (\r m -> (EnvT (\r1 -> ine r (applyEnvT m r1))))
          in LiftM {liftRdEnv = liftRdEnv_,
                   liftInEnv = liftInEnv_}
      in (typeChk l) top subtype newlf ot

```

The part of type checking the old expressions of the current building block is similar as that of `integerBlock`, except that besides making a new encoding of the subtyping relationship, we also need to make a new encoding of the lift operation to be passed as an argument to the lower level `typeChk` function. This is because the current building block extends the context monad with a layer of environment monad by applying a monad transformer to it, so the operations in the lower layer monads need to be lifted through this monad transformer. The local value `newlf` specifies the newly constructed encoding of the lift operation. To lift the lower level operations to the top level, we first lift them through the monad transformer `EnvT` to the current level, and then use the current level lift operator to lift them up to the top level.

4.3.3 A Building Block for Polymorphism

Finally, we show an interesting building block, which is for polymorphism. Before giving the definition, we state some important problems that do not show up in the former two building blocks but appear here. The

polymorphism building block will introduce two pieces of new expression syntax: explicit type abstraction and explicit type application. In the typical rule of type checking type abstraction expressions, we need to judge whether a type variable name appears freely in the type checking context; in the typical rule of type checking type application expressions, we need to do substitutions on type structures in order to instantiate a polymorphic type. So, the building block for polymorphism must have functionality to accomplish these two tasks. We add an auxiliary function `mapT` to the datatype of open systems to achieve this.

```
data Open exprGen typeGen envGen
  = Open {...
    mapT :: forall m t v. Monad m =>
        (t -> m v) ->
        (typeGen t -> m (typeGen v)),
    ...
  }
```

The function `mapT` is explicitly defined as polymorphic a function. This is the use of rank-2 polymorphism again. In its types, the universally quantified type variable `t` specifies the top-level type syntax structure; the type variable `m` specifies an arbitrary monad, as constrained by the type class predicate `Monad m`; the type variable `v` specifies the return type of the function to be mapped onto the type structure. One thing to note is that, the `mapT` function is not as usual map functions for algebraic datatypes. Normal map functions should have the type of the following form:

$$(a \rightarrow b) \rightarrow (C a \rightarrow C b),$$

where `C` is the datatype constructor. The `mapT` function has a type of the following form:

$$(a \rightarrow M b) \rightarrow (C a \rightarrow M (C b))$$

where `C` is still the datatype constructor, while `M` is a monad. So `mapT` actually maps a Kleisli function[6] onto an algebraic datatype. It does the mapping with a side effect encoded in a monad. If we take the monad as the identity monad, then the `mapT` function will be degraded to a normal map function. The side effect allows us to define many useful functions using `mapT`. As will be shown later, both the free type variable checking function and the type variable substitution function can be defined by using the facility offered here.

Now we can begin to show the definition of the building block for polymorphism.

```
polyBlock :: Open e t a -> Open (FExprGen <+> e) (FTypeGen <+> t) a
polyBlock l
  = let typeChk_ = ...
    in Open {typeChk = typeChk_}
```

The `polyBlock` extends an open system with new syntax structures of polymorphism. It does not extend the parameterized context monad, because there is no new expression variables syntax introduced by this building block.

The definition of the `typeChk` function for this building block also follows the basic strategy. We will only show a sketch of type checking the new expressions. The type checking of the old expressions is as same as that of `integerBlock`.

```

typeChk_top s _ (NewExpr nt)
  = let st      = ...                -- type substitution
      ftEnv     = ...                -- free type variable checking
      in case nt of
        TApp e t -> ... (st) ...    -- type check type application
        TLam t e -> ... (ftEnv) ...  -- type check type abstraction

```

Note that two functions, `st` and `ftEnv`, are used to do type substitution and free variable checking respectively. In the implementation of the typing rule of type application expressions, the function `st` is used to instantiate polymorphic types; in the implementation of the typing rule of type abstraction, the function `ftEnv` is used to compute the type variables that appear freely in the environment context. These two functions are both implemented using the auxiliary function `mapT`.

```

st (v, t) typ
  = case (proj s) typ of
    Just (NewType (TVar x))
      -> if x == v
          then return t
          else return typ
    Just (NewType (Forall x tt))
      -> if x == v
          then return typ
          else do y -> st (v, t) tt
                return (((inj s).NewType) (Forall x y))
    _ -> do x <- (topMapT top) (st (v, t)) typ
          return (MkType x)

```

In this code segment, `(proj s)` and `(inj s)` are the projection and injection functions encoded in the argument of the `typeChk` function that specifies the subtyping relationship between the current level type structure and the top-level type structure. The `topMapT` function is the counterpart of `mapT` in a closed system. It can be seen as the fix point of the `mapT` function of an open system. We use the `topMapT` function to recursively map the `st` function down to the whole structure of a type syntax in order to do a type substitution on it. Here the `mapT` function does not have any side effects. Note that there are no non-standard operations of the monad used in the definition of `st`. When put into use, the polymorphic monad in the type of `mapT` will be instantiated to an identity monad.

```

ftTyp typ
  = case (proj s) typ of
    Just (NewType (TVar v))
      -> do l -> getS
          putS (l 'setU' [v])
    Just (NewType (Forall v t))
      -> do ftTyp t
          l -> getS
          putS (l 'setDif' [v])
    _ -> do (topMapT top) ftTyp typ
          return ()

```

```
ftEnv a = ... ftType ...
```

The essence of `ftEnv` is the function `ftType` defined in this code segment. It collects all the type variables that appear freely in a type syntax structure. In defining `ftType`, the side effects of the `mapT` operator becomes important. During the process of traversing a type structure, we need to “remember” the free type variables that have been found. So the polymorphic monad is instantiated to a state monad. The `getS` function and `putS` function are the two non-standard operations of the state monad, where `getS` is used to read out the current state of the computation and `putS` is used to update the state. Note that the return value of `ftType` function becomes trivial (a void value is always returned). The result of this function focus on the side effect it produces.

4.4 Build a Closed System

So far we have finished the definitions of the three building blocks. In this section we will show how these building block can be composed to form a closed system with real checking facility. As mentioned, we need to have a special function `top` that transforms an open system to a closed one, and an initial open system `bot` as the base open system for the modular type system. Here we show the sketch of `top` and `bot`.

```
top :: ErrorMonad (c (Tfix t)) => Open e t c -> Closed e t c
top l
  = let typeChk_ (MkExpr t)
        = (typeChk l) (top l) ... t      -- compute the fix point of typeChk function
    in Closed {topTypeChk = typeChk_}

data BotExprGen e t = Unit                -- a single expression
data BotTypeGen t = Void                  -- a single type
data BotEnvGen t a = Ok a | Error         -- context monad is set to be an error monad
bot :: Open BotExprGen BotTypeGen BotEnvGen
bot = Open {typeChk = ...}
```

The function `top` takes an open system as argument and returns a closed system. The type class predicate `ErrorMonad (c (Tfix t))` restricts the context monad to have the ability of raising an error. The `topTypeChk` function field computes the fix point of the original open system’s `typeChk` function. So the function `top` shuts off the extension interface of an open system and generate a close system that supplies the type checking facility by the `topTypeChk` function field.

The value `bot` is an open system. It only contains one single expression and one single type, which is like the unit expression and unit type in Haskell. In this initial open system, we set the base of the context monad to be an error monad, so any open system built on the base of `bot` will be able to raise an error in its type checking context.

Now we can finally finish the construction of the static semantics of our modular type system that contains the type feature of integer arithmetic, λ -calculus and polymorphism:

```
my_ts = mkTS (integerBlock . lambdaBlock . polyBlock)
mkTS x = top (x bot)
```

4.5 Some Running Results

All the codes can be found at the URL:

http://www.cse.ogi.edu/~chiyan/mainpage/research_project/codes

These codes can be run under Hugs98 interpreter by loading the file `test.hs`. In `test.hs`, we define some values that represent type systems:

```
intSys    = mkTypeSys (integerBlock)
lambdaSys = mkTypeSys (integerBlock . lambdaBlock)
polySys   = mkTypeSys (integerBlock . lambdaBlock . polyBlock)
```

Some running results are shown as follows:

```
-- (tc intSys) "0"
"Int"

-- (tc intSys) "1 + (2 + 3)"
"Int"

-- (tc lambdaSys) "/x:Int -> x + 1"
"Int -> Int"

-- (tc lambdaSys) "(/x:Int -> x + 1) 2"
"Int"

-- (tc lambdaSys) "(2 1)"
"bad type!"

-- (tc polySys) "#a -> /x:a -> x"
forall a.(a -> a)

-- (tc polySys) "((#a -> /x:a -> x) [Int]) 1"
Int

-- (tc polySys) "#a -> /x:a -> x + 1"
"bad type!"
```

Here `tc` is the composition of the `topTypeChk` function field of the closed system and some parsing function that parses the character inputs to the abstract datatype of an expression closed syntax. The symbol `/` stands for the expression variable abstraction symbol λ . The symbol `#` stands for the type variable abstraction symbol Λ . All the running results are as expected.

5 Related Work

There are some other works related with the topic in this paper, such as Levin and Pierce's work of Tinker Type[7], and research on pure type systems[1]. However, they solve different problems from what we are dealing with here. Tinker Type focuses on modular specifications of type systems, while we focus on modular static semantics of type systems. In this sense, our work is more similar to Liang, Hudak and Jones' work of modular interpreters[8], except that their work is to explore the modularity of dynamic semantic of programming languages. The work of pure type systems defines a generic framework for type systems, allows the construction of type systems by switching parameters. It is different from our work because it does not build type systems in a compositional style, and does not divide the static semantics of type systems into independent and reusable building blocks as we do in this paper.

The essential idea of our method to decompose type systems is similar to Duponcheel[9] and Sheng Liang's work to decompose interpreters. However, we propose a totally different way of implementation in this paper. They make extensive use of constructor classes, while make use of rank-2 polymorphism.

The strength of their approach is that all the building blocks are define as an instance of a constructor class, so they can be implicitly composed by the type system and overloading mechanism of the language they use to do the implementation. However, they have to explicitly compose the open syntax pieces for expression structures and type structures. Also, they hide those subtyping relationships and lift operators for monad transformers using constructor classes. This makes their building blocks look elegant, without having those complex detail appearing explicitly. The price they pay is that all their building blocks depend on each other, so they must be organized in such a way that they are visible to anyone else. This prevent the building blocks from being separately compiled.

The strength of our approach is that we don't have to explicitly compose the open syntax pieces, because those types are strongly polymorphic so that they are implicitly composed when we do the composition of building blocks. However, we have to define explicit mechanism to compose the building blocks. Also, we don't rely on the implicit passing mechanism of constructor classes and all our subtyping relationships and lift operators are explicitly passed as function parameters, so our building blocks exist independently and can be separately compiled. The price we pay is that we lose the elegant presentation of building blocks, because the subtyping relationship and lift operators are explicit passed as function parameters, and all of their details need to be redefined in every building block repeatedly.

So it is about a trade-off here. Using their approach, it would be easier to add in dynamic features, because dynamic features can be encoded into monads and implemented using constructor classes. However, it will be inconvenient for them to add in a building block because that requires their overall implementation to be modified and recompiled. Using our approach, it would be easier to add in a building block, because all our building block exist independently and the newly added building block does not affect others at all. However, we meet trouble if we want to extends the system with some dynamic features. That would require radical changes to our overall framework because we need to add more parameters to the some functions

6 Conclusion and Future Work

In this paper we show a way to construct a modular type system by defining and composing reusable and independent building blocks. Although we only show three building blocks in this presentation, we actually make seven building blocks for the following type system features: integer arithmetic, logical expression, variable definition, first order functions, λ -calculus, Hindley-Milner features, and polymorphism. We make

a novel use of rank-2 polymorphism, which plays a important role in the framework of our modular type system; and we define a new kind of map operator that maps a Kleisli function onto an algebraic datatype, which makes the map operator more powerful.

This paper is only a starting attempt to construct modular type systems, and there are lots of opportunities for future work.

The framework needs to be refined. We are not satisfied with the means we currently use to deal with the subtype relation and the lift operator. We want to remove those explicit parameters from the interface of the type checking function to have a more elegant solution. This might involve the use of constructor classes[10].

The framework needs to be extended to accommodate new features. For example, if we want to include the feature of $\lambda\omega[1]$ into our modular type system framework, and thus a kind system, we must extend the framework to accommodate kinds besides expressions and types.

Also, we need to develop more building blocks in order to study the interaction between type system features, such as how qualified type[11] interferes with Hindley Milner polymorphism[12].

Another possible direction of this work is to develop higher order abstractions for modular type systems. We observe that there are similarities between some of the building blocks we have developed. We want to capture the similarities and design generic building blocks.

7 Appendix

7.1 The Definition of Constructor Class MonadGen

The constructor class `MonadGen` defines a class of parameterized monads. If `m` is a parameterized monad, then for any type `t`, `(m t)` is a monad.

```
class MonadGen m where
  returnG :: a -> m t a
  bindG   :: m t a -> (a -> m t b) -> m t b

instance MonadGen m => Monad (m t) where
  return = returnG
  (>>=) = bindG
```

7.2 The Definition of Datatype EnvT

The datatype `EnvT` is a parameterized monad transformer. It transforms a parameterized monad to another by adding a layer of environment monad feature.

```
data EnvT r m t a = EnvT (r t -> m t a)

applyEnvT (EnvT f) a = f a

instance MonadGen m => MonadGen (EnvT r m) where
  returnG x = EnvT (\r -> returnG x)
  m 'bindG' k
    = EnvT (\r -> applyEnvT m r 'bindG' (\x ->
      applyEnvT (k x) r))
```

Any `ErrorMonad` can be lifted through the `EnvT` transformer. If `(m t)` is an `ErrorMonad`, so is `(EnvT r m t)`.

```
instance (ErrorMonad (m t), Monad (EnvT r m t)) => ErrorMonad (EnvT r m t) where
  failE = EnvT (\r -> failE)
  liftE m = EnvT (\r -> liftE m)
```

References

- [1] Morten Heine B. Sorensen and Pawel Urzyczyn. Lectures on the Curry-Howard Isomorphism, 1998. DIKU Rapport, 98/14, URL: <http://www.cis.upenn.edu/~bcpierce/types/archives/current/msg00020.html>.
- [2] John C. Mitchell. Foundations for Programming Languages, 1996. Publisher: Mit Pr, ISBN: 0262133210.
- [3] Philip Wadler. The Essence of Functional Programming, 1992. Symposium on Principles of Programming Languages, pages 1-14.
- [4] Mark P. Jones. First-class Polymorphism with Type Inference, 1997. Symposium on Principles of Programming Languages, Pages 483 - 496.
- [5] John Launchbury. Monadic Semantics of a Domain-specific Language, 1997. Oregon Graduate Institute Technical Report, URL: <http://www.cse.ogi.edu/PacSoft/projects/SDRR/p2.report.html>.
- [6] Michael Barr and Charles Wells. Category Theory for Computing Science, 1990. ISBN: 2-921120-31-3.
- [7] Michael Y. Levin and Benjamin C. Pierce. Tinker type: a language for playing with formal systems, 1999. University of Pennsylvania Technical Report MS-CIS-99-19.
- [8] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters, 1995. Symposium on Principles of Programming Languages, pages 333-343.
- [9] Luc Duponcheel. Using Catamorphism, Subtypes and Monad Transformers for Writing Modular Functional Interpreters, 1997. Padualaan 14 3548 CH Utrecht The Netherlands.

- [10] Mark P. Jones. A System of Constructor Classes: Overloading and Implicit Higher-order Polymorphism, 1993. Functional Programming Languages and Computer Architecture, pages 52-64.
- [11] Mark P. Jones. A Theory of Qualified Types, 1992. European Symposium on Programming, pages 287-306.
- [12] Mark P. Jones. Formal Properties of the Hindley-Milner Type System, 1995. unpublsh manuscript.

APPENDIX E

Frappé: Functional Reactive Programming in Java*

Antony Courtney
Dept. of Computer Science
Yale University
New Haven, CT 06520
antony@apocalypse.org

Abstract

Functional Reactive Programming (FRP) is a declarative programming model for constructing interactive applications based on a continuous model of time. FRP programs are described in terms of *behaviors* (continuous, time-varying, reactive values), and *events* (conditions that occur at discrete points in time).

This paper presents Frappé, an implementation of FRP in the Java programming language. The primary contribution of Frappé is its integration of the FRP event/behavior model with the Java Beans event/property model. At the interface level, any Java Beans component may be used as a source or sink for the FRP event and behavior combinators. This provides a mechanism for extending Frappé with new kinds of I/O connections and allows FRP to be used as a high-level declarative model for composing applications from Java Beans components. At the implementation level, the Java Beans event model is used internally by Frappé to propagate FRP events and changes to FRP behaviors. This allows Frappé applications to be packaged as Java Beans components for use in other applications, and yields an implementation of FRP well-suited to the requirements of event-driven applications (such as graphical user interfaces).

1 Introduction

Recent work in the functional programming community has proposed Functional Reactive Programming (FRP) as a declarative programming model for constructing interactive applications. FRP programs are described in terms of *behaviors* (continuous, time-varying, reactive values), and *events* (conditions that occur at discrete points in time). The FRP style was first proposed in Fran [5], a domain-specific language for computer animation embedded in Haskell. The motivation for developing Fran was to enable the programmer to focus on *modelling* of the problem domain rather than on presentation details required to produce sequential frames of an animation. Since then, FRP has been adapted for numerous other application domains, including robotics [14, 13], vision [16] and graphical user interfaces [17].

All previous implementations of FRP have been embedded in the Haskell programming language [15]. As discussed in [8], Haskell's lazy evaluation, rich type system, and higher-order functions make it an excellent basis for development of new domain-specific languages and new programming paradigms such as FRP. Haskell has served as a basis for more than a half dozen implementations of FRP [4, 9, 17].

*This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

In the Java community, recent work has produced the Java Beans component model [2]. The Java Beans component model prescribes a set of programming conventions for writing re-usable "components" for use in an Integrated Development Environment (IDE) or other, similar tool (such as a web page authoring tool). A programmer writes a Java Beans component by defining a Java class that specifies a set of *events* ("interesting" conditions which result in notifying other objects of their occurrence) and *properties* (named mutable attributes of the component that may be read or written with appropriate methods). A visual builder tool uses Java's introspection facilities [11] to discover the events and properties exported by the component class. Many of the classes in the standard Java class libraries (such as those of AWT and Swing) are defined as Java Beans components.

The FRP and Java Beans programming models have very different goals and appear, at first glance, to be completely unrelated. The goal of FRP is to enable the programmer to write concise descriptions of interactive applications in a declarative modelling style, whereas the goal of Java Beans is to provide a component framework for visual builder tools. However, the two models also have some alluring similarities: both have a notion of *events*, and both have a notion of values that change over time (*behaviors* in FRP, *properties* in Java Beans). Our primary motivation for developing Frappé was to explore the relationship between the two models.

As a secondary motivation, we developed Frappé to explore using FRP for composing interactive applications from Java Beans components. All of the commercially available IDEs for Java allow the programmer to compose user interface layouts in a declarative, visual, direct manipulation style, but require the programmer to write (imperative) Java code to make this interface actually *do* anything. This approach raises a number of difficult issues for developers and users of IDEs, such as requiring every component of the UI to be exposed as a `public` field, ensuring that user-written Java code isn't lost when the UI layout code is regenerated, and managing all of the Java code attached to the UI to ensure that every event handler leaves the program in a consistent state. We believe that many of these problems could be redressed by specifying event handlers using some high-level declarative programming model instead of writing Java code directly. Providing an implementation of FRP in Java provides just such a model for Java application programmers and IDE developers.

This paper presents Frappé, an implementation of FRP in Java. Our implementation is based on a correspondence between the FRP and Java Beans programming models, and our implementation integrates the two models very closely. There are two aspects to this integration: First, any Java Beans component may be used as a source or sink for the FRP event and behavior combinators. Second, the Java Beans event model is used internally by Frappé for

propagation of FRP events and changes to FRP behaviors. Allowing any Java Beans component to be used as a source or sink for the FRP event and behavior combinators allows the Java programmer to use FRP as a high-level declarative model for composing interactive applications from Java Beans components, and allows Frappé to be extended with new kinds of I/O connections without modifying the Frappé implementation. Using the the Java Beans event model internally allows Java Beans components connected by FRP combinators to be packaged as larger Java Beans components for use by other Beans-aware Java tools, and yields a “push” model for propagation of behavior and event values that is well-suited to the requirements of graphical user interfaces.

The remainder of this paper is organized as follows. Section 2 gives a brief review of the FRP and Java Beans models. Section 3 describes an existing implementation of FRP in Haskell, and states some observations about the properties of this implementation. Section 4 presents our implementation of Frappé, using the observations presented in section 3 to justify the derivation, and also describes how Java Beans components can be used directly as sources or sinks for the FRP combinators. Section 5 summarizes the status of the implementation. Section 6 discusses some limitations of our implementation. Section 7 describes related work. Section 8 summarizes our contributions, and briefly discusses some open questions and plans for future work.

2 Preliminaries

2.1 Functional Reactive Programming

Functional Reactive Programming (FRP) is a high-level declarative programming model for constructing interactive applications. In this section, we give a very brief introduction to the aspects of FRP needed to understand the rest of the paper; see [5, 9] for more details. For concision, FRP examples presented in this section are given in Haskell [15], and, to keep the discussion grounded in concrete examples, many examples are taken from the problem domain of computer animation.

The principal idea that distinguishes FRP from most other programming models is that it presents a *continuous* model of time to the application programmer. Of course, actual implementations of FRP on real computers will only approximate the continuous model by some form of discrete sampling, but this implementation complexity is (mostly) hidden from the application programmer.

There are two key polymorphic data types in FRP: the *Behavior* and the *Event*. Conceptually, a *Behavior* is a time-varying continuous value. One can think of type *Behavior a* as having the Haskell definition:

```
type Behavior a = Time -> a
```

That is, a value of type *Behavior a* is a function from *Time* to *a*. Given this definition, we can think of sampling a behavior as simply applying the behavior to some sample time. The simplest examples of behaviors are *constant* behaviors: those that ignore their time argument and evaluate to some constant value. For example, `constB red` has type *Behavior Color*. It evaluates to `red` regardless of the sample time. An example of a time-varying behavior (taken from a binding of FRP for computer animation [5]) is `mouse` (of type *Behavior Point*). When sampled, `mouse` yields a representation of the mouse position at the given sample time. Sampling the mouse at different times may yield a different *Point* depending on whether the user has moved the mouse.

Conceptually, an *Event* is some condition that occurs at a discrete point in time. In Haskell, we write the type *Event a* for an *event source* capable of producing a sequence of *occurrences*, where each occurrence carries a value of type *a*. For example:

```
lbp :: Event ()
key :: Event Char
```

declare the types of two primitive event sources defined in the FRP binding for computer animation. The first event source, `lbp`, generates an event occurrence every time the left mouse button is pressed. Each occurrence carries a value of type `()` (read “unit”), meaning that there is no data carried with this event other than the fact that it occurred. The second event source, `key`, generates an event occurrence every time a key is pressed on the keyboard. Each occurrence carries a value of type `Char` representing the key that was pressed.

An implementation of FRP provides the programmer with a set of primitive behaviors and event sources, and a library of combinators for creating new behaviors and event sources from existing ones. For example, the expression:

```
lbp ==> red
```

uses the `==>` combinator (of type *Event a -> b -> Event b*) to produce an event source of type *Event Color*. The event occurs whenever `lbp` occurs (i.e. every time the left mouse button is pressed), but each occurrence carries the value `red`. More complex event sources and behaviors are produced by nested applications of combinators. For example, the expression:

```
(lbp ==> red .|. rbp ==> blue)
```

uses the *merge* operator (`.|. .`) to produce an event source (of type *Event Color*) that occurs whenever the left or right mouse button is pressed. When the left button is pressed, an occurrence is generated carrying the value `red`; when the right button is pressed, an occurrence is generated carrying the value `blue`.

The FRP model defines a combinator, *switcher*, for converting an event source to a behavior. The type of *switcher* is given as:

```
switcher :: Behavior a -> Event (Behavior a) -> Behavior a
```

Informally, *switcher* produces a behavior that initially follows its first argument. However, every time an event occurs on the event source given as the second argument, *switcher* “switches” to follow the behavior carried in that event occurrence. For example:

```
c = switcher red (lbp ==> red .|. rbp ==> blue)
```

uses *switcher* to define `c` as a behavior with type *Behavior Color*¹. When the application starts, `c` will initially be `red`. When the left mouse button is pressed, `c` changes to `red`, and when the right mouse button is pressed, `c` changes to `blue`.

Complete applications are written in FRP by defining *reactive* behaviors. Reactive behaviors are behaviors that change in response to user input, such as the definition of `c` just given. A binding of FRP for a particular problem domain will usually define a type for a top-level behavior that represents the output of the application. A complete application is defined simply by using the FRP combinators to write an expression for a behavior of this type. For example, in computer animation, the output of an application is of type *Behavior Picture*. An example, then, of a complete FRP application for computer animation is:

```
exampleApp = withColor c circle
  where c = switcher red (lbp ==> red .|.
                        rbp ==> blue)
```

¹Here we use *implicit lifting* of constants to Behaviors. Strictly speaking, we should have written `constB red` instead of just `red`, but the Haskell implementations of FRP use *instance declarations* to perform this translation automatically.

this application renders a circle of unit size in an output window. The circle will be initially red, but the color will change between red and blue as the left and right mouse button are pressed.

2.2 Java Beans

This section gives a brief summary of the Java Beans programming model. For a more complete account, the reader is referred to the Java Beans Specification [2].

2.2.1 What is a Bean?

The Java Beans Specification defines a Java Bean informally as “a reusable software component that can be manipulated visually in a builder tool”.² For example, all of the Swing user interface components (buttons, sliders, windows, etc.) are Beans. However, Beans need not have any visual appearance at run-time. For example, a software component that takes a ticker symbol as input and periodically delivers stock quotes for the ticker symbol could easily be packaged as a Bean.

More concretely, then, a Bean is a Java class that conforms to certain programming conventions prescribed by the Java Beans specification. These conventions specify that a Bean should make its functionality available to clients through:

- *Properties*—mutable named attributes of the object that can be read or written by calling appropriate *accessor* methods.
- *Events*—A named set of “interesting” things that may happen to the Bean instance. Clients may register to be notified when an event occurs by implementing a *listener* interface. When the event occurs, the component notifies the client by invoking a method defined in the listener interface.
- *Methods*—Ordinary Java methods, invoked for their side-effects on the Bean instance or its environment.

The Beans model does not require a separate interface definition language for specifying the interface to a Java Beans component. Instead, the Beans model prescribes that a Java Bean can be written as an ordinary Java class in the Java language, and can be packaged for use by a builder tool simply by compiling the source file to the standard Java class file format. The “builder tools” use reflection [11] on the class file to recover information about the features exported by the particular Bean, and the standard Java library provides a set of helper classes in the `java.beans` package for use by builder tools. These helper classes perform the low-level reflection operations to recover information about events, properties and methods supported by a Bean.

2.2.2 Properties

Properties are mutable, named attributes of a component. For example, each of the graphical user interface components in the Java Swing library defines *width* and *height* properties that represent the bounding box of the component’s visual representation. Conceptually, a property is similar to a field of an object, and properties are often implemented by storing the property value in a field. However, a property need not be implemented this way; a component implementation could instead compute a property’s value dynamically when the property is read.

Properties are accessed by means of *accessor methods*, which are used to read and write values of the property. A property may

be *readable* or *writable* (or both), which determines whether the property supports *get* or *set* accessor methods. The Java Beans specification defines programming conventions that component authors must follow when writing accessor methods. The convention for *get* accessor methods is:

```
public PropertyType getPropertyName();
```

and the convention for *set* accessor methods is:

```
public void setPropertyName(PropertyType arg);
```

where *PropertyName* is the (appropriately capitalized) name of the property, and *PropertyType* is the Java type of the property. Continuing with the previous example, the class `JComponent` of Java Swing defines *get* and *set* accessors for its *width* and *height* properties as:

```
public class JComponent {  
    ...  
    public int getWidth();  
    public void setWidth(int arg);  
  
    public int getHeight();  
    public void setHeight(int arg);  
}
```

Invoking `getWidth()` will return the width of the component (in pixels); Invoking `setWidth()` will set the component’s width (in pixels).

2.2.3 Events

Conceptually, *events* in the Java Beans component model are a mechanism used by a *source* component to notify one or more target *listeners* of some state change in the source object. For example, a `Button` user interface object might define an *action* event that is used to notify listeners when the button has been pressed by the user. The Java Beans specification requires component authors to adhere to the following conventions:

- The programmer must specify an *event* class to encapsulate the information about a single event occurrence. This event class should be a subclass of `java.beans.Event` named *event-nameEvent*, where *event-name* is a logical name for the event chosen by the component author.³
- The programmer must specify a *listener* interface, that defines one or more *notification* methods. A listener interface has the form:

```
public interface event-nameListener {  
    public void notifyMethod(event-nameEvent arg);  
    ...  
}
```

where *event-name* is as before, and *notifyMethod* is a method name (chosen by the component author) to be invoked when the event occurs.

- The programmer must add a pair of *listener registration* methods to the component acting as the event source. These listener registration methods have the form:

³Technically, *event-name* identifies a *set* of related events rather than a single event, but this distinction is not important here.

²In the remainder this paper, we use the terms “Java Beans component” and “Bean” interchangeably.

```

public void
  addEventListener(event-nameListener lis-
    tener);

public void
  removeEventListener(event-
    nameListener listener);

```

The former method registers a listener object to be notified when the given event occurs; the latter removes a previously registered listener.

An example of a Java Beans event is the *action* event defined by the `JButton` class from Java Swing. An application arranges to be notified when a `JButton` is pressed by implementing the `ActionListener` interface and passing a reference to an instance of this interface to the `addActionListener()` method of `JButton`. The implementation of `JButton` records the listener registration in its listener list. Later, when the button is pressed by the user, the `JButton` notifies each registered listener by invoking the listener's `actionPerformed()` method, passing it a reference to an `ActionEvent` with information about the event occurrence.

2.2.4 Bound Properties

A particularly important aspect of the Java Beans specification is its provision for *bound* properties. A component author may specify that a property is a *bound* property, meaning that interested clients can register to be notified whenever the property's value changes. A property's value might change either as a direct result of the application program invoking a *set* accessor method, or as an indirect result of some user action. For example, a text entry widget might have a *text* property representing the text entered into the field by the user. If implemented as a bound property, an application could register to be notified whenever the user changed the contents of the text entry component.

The Java Beans specification defines a `PropertyChangeListener` interface that carries information about a change to a bound property (such as its old value and new value), and a `PropertyChangeListener` interface that must be implemented by client objects that wish to be notified of changes. As we will see shortly, bound properties play a critical role in the implementation of FRP Behaviors in Frappé.

3 Analysis of Stream-Based FRP

To date, there have been numerous implementations of FRP in Haskell. Some of the design alternatives (and their associated engineering tradeoffs) are explored in [4]. An efficient, robust implementation of the FRP model that is both faithful to the formal semantics and does not suffer from space leaks or other performance problems has remained an elusive goal, and we do not expect that this situation will change in the near future.

One recent implementation of FRP that has received particular scrutiny is the so-called stream-based implementation, described in [9].⁴ A related paper [18] analyzes this implementation formally, showing that, with one caveat⁵, as the sample time approaches 0, the implementation is faithful to the formal semantics.

We present our design by first describing the stream-based FRP implementation in Haskell. Then we state some simple, informal observations about the stream-based implementation. In the next

⁴In fact, there have been multiple independent stream-based implementations of FRP. In this paper, "the stream-based implementation (of FRP)" refers only to the implementation by Hudak [9].

⁵Namely, that the implementation is unable to detect instantaneous predicate events. This issue is discussed more fully in section 6.

section, we will use these observations to derive an event-driven implementation of FRP in Java.

3.1 Stream-Based Implementation of FRP

The stream-based implementation defines FRP Behaviors and Events as *stream transformers* (in the signal processing sense) with the following Haskell definitions⁶:

```

type Behavior a = [(Maybe UA,Time)] -> [a]
type Event a = [(Maybe UA,Time)] -> [Maybe a]

```

Here `UA` is a type that represents a user action as obtained from the Operating System, such as a mouse event or a key being pressed. The type `Maybe UA` is a *possible* user action; its value is either `Nothing` or `Just x`, where `x` is some `UA`. Another important detail is that because Haskell uses lazy evaluation, lists and streams have exactly the same representation. Hence, for our purposes, we can read `[a]` as "stream of `a`".

A `Behavior`, then, takes an input stream consisting of pairs of possible user actions and sample times, and produces an output stream of behavior values. An `Event` source takes the same input stream as a `Behavior`, but produces an output stream consisting of *possible* event occurrences.

The FRP model defines a set of combinators (such as `switcher`, `-->`, etc.) for composing new behaviors and events from existing ones. An FRP program is an expression composed from nested applications of these combinators to other combinators or to primitive behaviors and events provided by the implementation. The stream-based implementation (and all of the other Haskell implementations, for that matter) implement these combinators directly as Haskell functions. However, as with all Haskell programs, Haskell's lazy evaluation semantics result in the implicit construction of a *graph* at runtime, where each node in the graph is an FRP combinator, and each edge represents the application of that combinator to some other `Behavior` or `Event`. This is illustrated in figure 1.

This figure illustrates a stream-based implementation of the FRP model for computer animation⁷. The user program is a graph structure, the inputs to the graph are streams of sample times and user actions, and the output of the graph is a stream of `Picture` values (one for each sample time fed as input). Because of lazy evaluation, control flow starts at the outputs, and successive `Picture` values are "pulled" from the graph by the application of `MapM.drawPic` on the right to the output of the user program. Applying `MapM.drawPic` to the output stream forces the computation of the first `Picture` of the output stream (i.e. the head of the list). To compute this picture, the outermost combinator node in the graph of the user program must compute a value. Typically this will force the computation of the inputs to this outermost node (i.e. the combinator nodes to which this node was applied when defining the user program.)

Control flow proceeds in this way (from outputs to inputs of each node) until some combinator in the graph forces the computation of the next `(Time,UserAction)` pair from the input stream. When this happens, the implementation will make one blocking call to the Operating System's event loop, returning only when either a user action event has occurred or one sample interval has passed. In either case, the implementation provides values for the heads of the input streams of the graph, and control flow works its way back through the graph until the computation of the current `Picture` value is complete. The `Picture` value is then rendered on

⁶Up to an isomorphism. The actual types have been massaged slightly to simplify the exposition.

⁷Hence the type of user program as `Behavior Picture`.

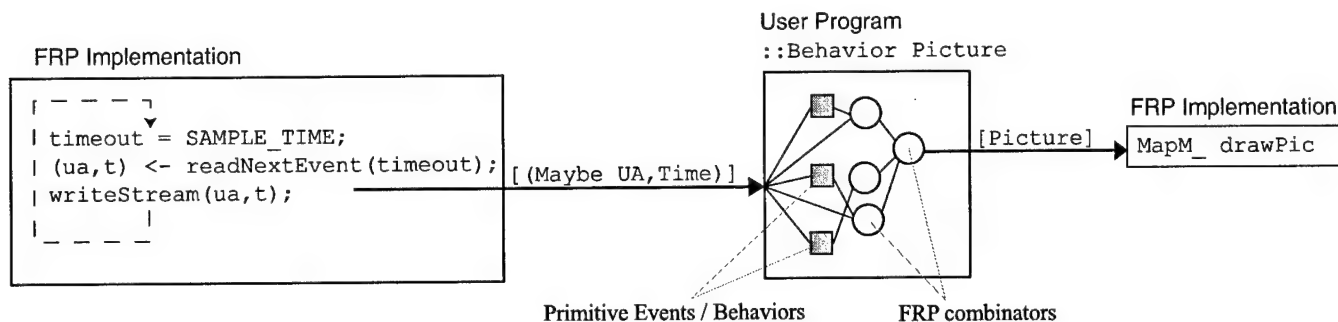


Figure 1: Stream-based implementation of FRP

the screen by `drawPic`, `MapM` forces the computation of the next `Picture` value of the output stream, and the whole process repeats.

3.2 Observations

From the preceding description, we can make the following observations about the stream-based implementation:

1. *FRP Programs are really directed graphs at runtime.*⁸ Each node in this graph corresponds to an FRP combinator, and each edge corresponds to an application of the combinator to some other combinator or a primitive event or behavior. The graph is constructed implicitly as a by-product of lazy evaluation, and is completely hidden from the user, but the graph structure *must* exist in some form at runtime to provide accurate lazy evaluation semantics.
2. *Sample times increase monotonically.* Each time a new `Picture` value is computed, it results in reading the next `UserAction` or `Time` value from the program's input stream, which in turn results in a single pass through the operating system's event loop.

Regardless of whether the call to the event loop returns a `UserAction` or times out, the `Time` value passed to the program's input stream is greater than or equal to the time on the input stream when the previous frame was computed. Most of the FRP combinators in the implementation depend on this monotonicity property.

3. *Each FRP combinator only examines the head of the input stream.* There are two requirements on the implementation that validate this observation. The first is that FRP combinators do not retain any history of previous `(Maybe UA, Time)` value pairs. If an FRP combinator did retain such values, this could lead to a space leak, as described in [4].

The second requirement is that each combinator must produce a value on its output stream every time a new `(Maybe UA, Time)` value is delivered on its input stream. This latter requirement is necessary because each `(Maybe UA, Time)` value demanded from the input stream results in a blocking call to the operating system's event loop. If any FRP combinator demanded another value from its input stream before producing any output, then, in the presence of nested and recursive combinators, this could result in arbitrarily long delays before the program produced any output.

⁸This is true of any Haskell program, but is critical to understanding the sequence of actions in the stream-based FRP implementation.

4. *The stream-based implementation performs sampling even for purely event-driven applications.* Consider the following Behavior:

```
mycolor :: Behavior Color
mycolor = red 'until' (lbp ->> blue)
```

This defines a behavior (of type `Color`) that will be `red` until the left mouse button is pressed, and then will change to `blue`. Behaviors such as this one (common in graphical user interfaces) are purely *event-driven* in the sense that, while their value changes over time, their value depends solely on user input events, not on the sample time. Contrast this with a definition such as:

```
wiggle :: Behavior float
wiggle = sin time
```

which is an example of a *time-dependent* behavior.

In the stream-based implementation of FRP, the implementation performs sampling regardless of whether or not the FRP program actually has any time-dependent behaviors. That is, even if the user program is purely event-driven, the implementation will still produce a new `(Maybe UA, Time)` pair on the program's input stream every sample interval, and compute a value to place on the output stream of every combinator in the graph. We refer to this as the "sampling overhead" of the stream-based implementation.

4 Implementing FRP in Java

In this section, we present our implementation of FRP in Java. We use the observations about the stream-based implementation presented in the previous section to justify the validity of our design. Since the derivation is based on the observations about the stream-based implementation, and since the stream-based implementation has been shown to implement the formal semantics of FRP, we claim that our event-driven design is also faithful to the formal semantics. With a suitable framework for formalizing the observations in the last section, and for describing the Java constructs used in our implementation, it should be possible to give an operational semantics of Frappé in terms of the stream-based implementation of FRP in Haskell.

4.1 Behaviors

Recall the first three observations of section 3.2:

1. FRP Programs are really directed graphs at runtime.

```

public interface Behavior {
    /** Accessor to read the current value of this Behavior
     */
    public Object getValue();

    /** Add a PropertyChangeListener to be notified when
     * the value of this Behavior changes.
     */
    public void
        addPropertyChangeListener(PropertyChangeListener l);

    /** Remove a PropertyChangeListener from the list of listeners.
     */
    public void
        removePropertyChangeListener(PropertyChangeListener l);
}

```

Figure 2: Java encoding of Behaviors

2. Sample times increase monotonically.
3. Each FRP combinator only examines the head of the input stream.

Since Java does not have lazy evaluation, we must construct the graph of combinator applications explicitly. We achieve this by defining a Java class for each FRP combinator. Each node in the combinator graph is represented by an object instance at runtime, and each edge is represented by a field with a reference to another object.

What operations must each node in the combinator graph support? We can combine observations (2) and (3) above to reach a somewhat surprising conclusion:

- Each Behavior node in the FRP combinator graph can be modelled as an object that has just one operation: get the value of the Behavior at the current time.

We can implement this model by defining a Behavior as a Java Bean with a single, readable property (called “value”). The Behavior’s value at the current time is obtained by reading the Behavior’s “value” property.

Individual Behavior objects might be connected as inputs to other nodes in the combinator graph, and those nodes will need to be informed when a Behavior’s value has changed. Hence, we make *value* a *bound* property, so that other nodes can register for a *PropertyChangeEvent* when the value of the Behavior changes. Our implementation uses such events to propagate behavior values through the system.

This leads to the Java encoding illustrated in figure 2. While the syntax is somewhat verbose, this can be read simply as “Every Behavior is a Bean that provides a bound property named *value*.”

An implementation of the Behavior interface supports registration of *listeners*, as required of bound properties. All output connections for a node are stored in this listener list. If an FRP combinator class uses a Behavior as one of its inputs, the combinator class must implement the *PropertyChangeListener* interface in order to be notified of changes in its input behavior. We will see an example of this shortly.

Since the Haskell definition of Behavior is a polymorphic type, we declare the return type of *getValue()* as *Object*. The value returned must be converted to an instance of the appropriate type using a cast, and the cast will be checked at runtime. This is quite inefficient for behaviors of primitive types, since each value of a primitive type must be wrapped in an instance of the appropriate wrapper class (*Integer*, *Float*, etc.) before it can be returned as an *Object*. To help avoid this inefficiency, and to provide

rudimentary static type checking, we have defined a set of sub-interfaces that specialize Behavior at all of the primitive types by providing an accessor method specialized to the primitive type. For example:

```

public interface FloatB extends Behavior {
    public float getFloatValue();
}

```

is the specialization of Behavior at the primitive type *float*. If a client of the Behavior interface knows the type of a Behavior’s value ahead of time, specialized interfaces like *FloatB* can be used instead of Behavior to avoid creating the intermediary wrapper object and the corresponding runtime type check.

4.2 Events

We implement FRP Events by mapping the FRP notion of “event” directly to a Bean Event named *FRPEvent*. Frappé defines one class and two interfaces for event handling, shown in figure 3.

The class *FRPEvent* represents a single event occurrence. It extends *java.util.EventObject*, as required by the Java Beans specification. As defined here, the only information carried with an event occurrence is a reference to the *FRPEventSource* that generated the event. In the FRP model, however, each event occurrence may carry a data value of some type. We accommodate this by defining a subclass of *FRPEvent* called *ObjectEvent* that carries a single value (of type *Object*). For convenience, we also define several subclasses of *ObjectEvent* specialized at various types. For example, we define a *BehaviorEvent* class for events that carry a Behavior reference on each occurrence.

The *FRPEventSource* interface is implemented by every class that generates FRP Events. This interface corresponds directly with the Haskell type *Event a* that identifies an event source in the stream-based implementation. The methods defined in *FRPEventSource* are those prescribed by the Java Beans conventions for registering event listeners. This interface declaration can be read as stating that “Every FRP Event Source is a Bean event source for the event named *FRPEvent*.”

The *FRPEventListener* interface is implemented by any class that wishes to be notified when an *FRPEvent* occurs on some source. A listener is registered with the event source by passing a reference to the listener to the source’s *addFRPEventListener()* method. Then, at some point later when the event occurs, the event source will notify all registered listeners by invoking each listener’s *eventOccurred()* method, passing it an *FRPEvent* instance representing the event occurrence.


```

/** Class representing a single event occurrence */
public class FRPEvent extends EventObject {
    public FRPEvent(FRPEventSource source) {
        ...
    }
}

/** Interface implemented by interested listeners */
public interface FRPEventListener extends EventListener {
    /** Invoked when the given FRPEvent has occurred */
    public void eventOccurred(FRPEvent event);
}

/** Source of FRP events; provides methods for registering
 * listeners to be notified of occurrences.
 */
public interface FRPEventSource {
    public void addFRPEventListener(FRPEventListener listener);
    public void removeFRPEventListener(FRPEventListener listener);
}

```

Figure 3: Java Encoding of FRP Events

4.3 Defining Combinators

In our implementation, every FRP combinator is implemented as a Java class. Each combinator class implements one of the standard FRP interfaces (Behavior or FRPEventSource) appropriate to the result type of the combinator. The arguments to the combinator are typically passed as arguments to the constructor.

For example, in Haskell, we might define an `overPic` combinator for compositing two animations (of type Behavior Picture, or, more succinctly, PictureB) into a single animation. In Haskell, this combinator would have the following signature:

```
overPic :: PictureB -> PictureB -> PictureB
```

In our Java implementation, we would implement this as:

```

public class OverPic
    implements Behavior,
        PropertyChangeListener {
    public OverPic(Behavior pic1,
                  Behavior pic2)
        ...

    // invoked when pic1 or pic2 changes:
    public void propertyChanged(...) {
        ...
    }

    // get the composition of pic1 and pic2:
    public Object getValue(...) {
        ...
    }

    // not shown: addPropertyChangeListener,
    //             removePropertyChangeListener.
}

```

Note that `OverPic` implements two interfaces: `Behavior` (since the combinator produces a Behavior) and `PropertyChangeListener`. This latter interface, defined in the Java Beans Specification, requires that the object implement the `propertyChanged()` method. Implementing this interface allows `OverPic` to register itself as a `PropertyChangeListener` on `pic1` and `pic2`, which will invoke `OverPic`'s `propertyChanged()` method when either picture changes.

Of course, as noted, this encoding requires the programmer to pass Behaviors that carry values of the appropriate type (Picture)

to `OverPic`. Otherwise a runtime error will result when `OverPic` attempts to cast the result of, say, `pic1.getValue()` to type `Picture`.

4.4 Propagation of Event and Behavior Values

Propagation of event and behavior values in Frappé is purely event-driven. To execute an FRP specification, a user program simply constructs an explicit graph of FRP combinators (*initialization*), and relinquishes control to the main event loop in the Java runtime library. When there is input to the application (for example, when the user presses a mouse button), the Java runtime will invoke an event handler of some object in the Frappé implementation that implements a primitive FRP event source or behavior. This primitive event handler, in turn, will invoke the appropriate event handler of each registered listener:

- For an event source, each event listener implements the `FRPEventListener` interface. The listener's `eventOccurred()` method is invoked to propagate the event occurrence.
- For a behavior, each event listener implements the `PropertyChangeListener` interface. The listener's `propertyChanged()` method is invoked to propagate the change in the behavior's value.

Each registered listener for a primitive event or behavior is an FRP combinator. The combinator's event handler will compute any changes to its output and invoke an event handler method on each of its registered listeners. Propagation of events continues in this way until some "output" listener is reached.

(Maybe have a diagram here??)

4.5 Where did the Time Go?

In the stream-based implementation, every Behavior has access to an input stream of user actions and sample times. However, as observation (3) of section 3.2 makes clear, the only time value that is actually accessible to a combinator from this input stream is the *current time*. Of course, this could just as easily be accessed from the global Behavior `time` defined by the FRP implementation.

Like other FRP implementations, Frappé provides user programs with a Behavior that represents the current time:

```

public class Animator {
    ...
    public FloatB getTimeInstance();
}

```

The behavior returned from `getTimeInstance()` is equivalent to the global Behavior `time` in other Haskell-based implementations. This primitive behavior is implemented by a software timer (our implementation uses an instance of `javax.swing.Timer` internally). The implementation maintains an internal floating point value that represents the current time (in seconds) since the application started. The software timer causes an event handler on the time implementation to be invoked at regular sample intervals. Each time this event handler is invoked it updates its notion of the current time, and propagates a `PropertyChangeEvent` to all registered listeners.

Our treatment of `time` differs from the previous Haskell-based implementations in that combinators that need access to the time instance must have the time instance passed in explicitly. To illustrate this difference, consider, for example, an *integrator* that implements a numerical approximation of the mathematical function defined as

$$\text{integrator}(v, t) = \int_0^t v(t)$$

The stream-based implementation (as well as other Haskell-based implementations) define a combinator `integral` with the Haskell signature:

```

integral :: FloatB -> FloatB

```

The expression `integral x`, when applied to some `x` (of type `FloatB`) yields another `FloatB` equal to `integrator(x, time)`, where `time` is the current time. That is, the stream-based implementation does not require that `time` be passed in explicitly to `integral`.

Our implementation also provides approximate numerical integration. However, we require that the `time` instance be passed in explicitly:

```

FloatB x, time, ix;
...
time = animator.getTimeInstance();
ix = new Integral(x, time);

```

Requiring that `time` be passed explicitly enables an important optimization: the implementation can eliminate the “sampling overhead” of the stream-based implementation described in section 3. All combinators that are time dependent must be explicitly passed the `time` instance returned from `Animator.getTimeInstance()`. Such time-dependent combinators will then register to be notified every time the sample time changes by calling the `addPropertyChangeListener()` method of `time`. If an application is purely event driven, then there will be no registered listeners on the `time` instance. In this case, the Frappé implementation turns off the software timer used internally, which allows the Java runtime to perform an indefinite (i.e. no timeout) blocking call to the operating system’s event loop, putting the application to sleep until some user action occurs.

This treatment of time has an important effect on how time transformations are handled in Frappé. The Haskell-based implementations of FRP provide a generalized time transformation combinator for transformation of local time-frames:

```

timeTransform :: Behavior a -> TimeB -> Behavior a

```

This could be used to transform a Behavior *without knowing the time-dependencies* of the behavior. For example:

```

timeTransform x (time/2)

```

produces a version of `x` that is slowed by a factor of 2, regardless of how `x` is defined.

In Frappé, it is still possible to perform time transforms, albeit in a substantially limited form. The programmer may only transform time-dependent behaviors, and must do so by interposing the appropriate transformation between the combinator and its time source. So, for example, to speed up an animation by a factor of two we must replace:

```

b = new BouncingBall(time);

```

with:

```

b = new BouncingBall(new DoubleArg(time));

```

where `DoubleArg` represents some combinator that doubles its argument behavior.

Since our encoding of FRP events does not explicitly time stamp each event, there is no direct mechanism for applying time transforms to event sources or to behaviors that are driven by event sources other than time.

4.6 Connecting to Java Beans

The stream-based implementation of FRP defines a fixed set of “primitive” behaviors and events that are available to user programs. These primitives typically represent I/O connections available to the application. For example, `mouse` is a primitive behavior corresponding to the current mouse position, and `lbp` is a primitive event source whose event occurrences correspond to the left mouse button being pressed. Because this set of primitives is implemented by interfacing directly with the Operating System, adding a new kind of I/O connection to the FRP implementation requires source-level changes to the FRP implementation.

In contrast, Frappé allows any Java Bean to be used as a source or sink for the FRP combinators. Recall from section 4.1 that a Behavior is just a Java Bean with a bound property named `value`, of type `Object`. We can treat any bound property of any Java Bean as a Behavior simply by constructing a Behavior that changes whenever the named property of the Bean changes. We provide a `PropertyObserver` class for this purpose:

```

public class PropertyObserver
    implements Behavior,
        PropertyChangeListener {

    public PropertyObserver(Object target,
                           String propName) {
        ...
    }
    ...
}

```

`PropertyObserver` uses reflection on the target object to obtain Method objects that represent the target’s `addPropertyChangeListener()` and `getpropName()` methods. The former is used in the constructor of `PropertyObserver` so that the observer instance will be notified when the target property changes, and can propagate the change to its registered listeners. The latter is used in the implementation of `getValue()` in `PropertyObserver`, so that all invocations of `getValue()` on the observer are forwarded to the appropriate accessor method of the named property on the target object.

On the output side, we have defined a similar utility class for connecting a Behavior to a writable bean property:


```

public class BehaviorConnector
    implements PropertyChangeListener {

    BehaviorConnector(Behavior src,
                     Object target,
                     String propName) {
        ...
    }
}

```

A `BehaviorConnector` registers itself as a `PropertyChangeListener` on the source behavior `src`, and uses reflection to look up the write method for the property name `propName` on the Bean `target`. Every time the source behavior changes, the `BehaviorConnector` instance obtains the value of `src` and uses the write method to set the value of the named property of `target`.

5 Current Status

We have working implementations of all of the core combinators given in [12], using the encoding of behaviors and events presented in the preceding section. For the most part, the implementation of these combinators is a straightforward translation of the formal definition into the Java language using the types and propagation model presented here. The code for a prototypical example combinator (`Switcher`) is given in Appendix A.

We have tested our implementation by rewriting many of the examples from [12] in Java. The code for a typical example is given in Appendix B. Our experience thus far is that the library is quite easy to use, but (unsurprisingly) the syntax for composing FRP specifications is exceedingly verbose compared to the embedding of FRP in Haskell. However, since we plan to use Frappé primarily as the back-end for a higher-level visual composition tool, this syntactic overhead is not too much of a concern.

6 Limitations

Because Java lacks a polymorphic type system, our implementation of FRP is not statically type-safe. Since we plan to use our implementation primarily as a compilation target for some other high-level tool, this is not too much of a concern; we expect tools that generate Frappé combinator graphs to be aware of the polymorphic nature of the FRP model, and to only generate combinator graphs that will not have type errors at runtime. Nevertheless, it would be an interesting exercise to rewrite Frappé using GJ [1].

Frappé assumes that event processing is single-threaded and synchronous. That is, all primitive Java Beans events used as event or behavior sources for Frappé must be fired from the system's event dispatching thread, and each event must completely propagate through the FRP combinator graph before the next event is handled. This single-threaded, synchronous event processing model is also required by Java Swing, and Frappé does not impose any further restrictions than those already required for event handling in Swing.

Like the stream-based implementation from which it derives, our implementation of FRP is unable to detect instantaneous predicate events. An instantaneous predicate event is one that happens only at some specific instantaneous point in time. For example:

```

sharp :: Event ()
sharp = when (time==1)

```

is only true instantaneously at `time=1`. An event such as `sharp` can not be detected simply by monotonic sampling; accurate detection of predicate events requires *interval analysis*, as discussed in [5, 4].

In many ways, the inability to detect instantaneous predicate events is similar to the problem of comparing two floating point numbers for equality using `==`, lifted to the time domain.

Finally, as discussed in section 4.5, Frappé only provides an extremely limited form of time transformation, and this limited form of time transformation violates the principle of *temporal modularity* proposed for Fran [5]. While time transform is undoubtedly useful for the specific domain of computer animation, it is less clear how useful generalized time transforms are when using FRP for other problem domains (such as graphical user interfaces).

We feel that there has simply not been enough experience using FRP to write real applications to know how significant these latter two limitations are in practice.

7 Related Work

Elliot [4] has done much of the pioneering work on implementations of the FRP model in Haskell, and reported on the design tradeoffs of various implementation strategies. Hudak [9] provides a completely annotated description of a stream-based implementation of FRP from which our implementation is derived.

Recent work in the functional programming community has produced ways to make component objects and library code written in imperative languages available from Haskell [7, 6, 10]. Our work and this previous work share the common goal of providing programmers with a declarative model for connecting component objects written in imperative languages. However, our approach can be viewed as the “inverse” of these efforts: instead of embedding calls to component objects written in an imperative language into a declarative programming model, Frappé takes a declarative programming model and embeds it in an imperative language that supports component objects.

Elliot's work on *declarative event-oriented programming* [3] showed that FRP's event model (implemented in Fran) could be used to compose interactive event-driven user interfaces in a declarative style, and compared this to the conventional imperative approaches for programming user interfaces. `FranTk` [17] is a complete binding of the FRP programming model to the Tk user interface toolkit. `FranTk` demonstrates the viability of using FRP for user interfaces, and inspired us to explore how we might adapt the FRP model for use with the Java Swing toolkit.

8 Conclusions and Future Work

We have presented an implementation of FRP in the Java programming language. The most significant aspect of our implementation is that it is based on a close correspondence between the FRP event/behavior model and the Java Beans event/property model. Our implementation imposes some limitations on the FRP model, but preserves FRP's declarative nature and continuous model of time.

Our motivation for implementing Frappé was to make the rich FRP model available to Java programmers and, in particular, to IDE implementers. We believe that, with a suitable visual notation, it should be possible to substantially improve on existing user interface builder tools included in most IDEs. We are currently developing a new visual application builder tool on top of Frappé to explore this hypothesis. This tool allows the user to compose entire applications (not just user interface layout) in a visual, declarative, direct manipulation style, by connecting individual user interface components through a diagrammatic representation of FRP combinators. The tool generates Java code to instantiate the user interface components and the Frappé objects connecting those components.

As stated in section 6, our current implementation of Frappé imposes some limitations on the FRP model. An interesting area for future research will be examination of whether these limitations can be removed in the Java implementation.

One of the unique aspects of Frappé is its ability to use Java Beans components as sources or sinks for FRP combinators. In principle there is no reason why this features needs to be limited to our Java implementation of FRP. It would be interesting to explore adding a similar feature to one of the Haskell-based implementations of FRP using COM objects as components.

9 Acknowledgements

John Peterson provided invaluable feedback on early draft of this paper. Paul Hudak has provided encouragement, support and suggestions as this work has progressed. We are grateful to Valery Trifonov and Zhanyong Wan for discussion and debate on the limits of the implementation presented here.

References

- [1] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, Oct. 1998.
- [2] G. H. (editor). *Java Beans API Specification 1.01*. Sun Microsystems, 1997. Available online at <http://java.sun.com/beans/>.
- [3] C. Elliott. Declarative event-oriented programming. Technical Report MSR-TR-98-24, Microsoft Research, October 1998.
- [4] C. Elliott. Functional Implementations of Continuous Modelled Animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.
- [5] C. Elliott and P. Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
- [6] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34.9 of *ACM Sigplan Notices*, pages 114–125, N.Y., Sept. 27–29 1999. ACM Press.
- [7] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. *H/Direct*: A binary foreign language interface for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 153–162. ACM, June 1999.
- [8] P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.
- [9] P. Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, Cambridge, UK, 2000.
- [10] S. P. Jones, E. Meijer, and D. Leijen. Scripting COM components in haskell. In *Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998.
- [11] S. Microsystems. *Java Core Reflection API and Specification*. Sun Microsystems, 1997. Available online at <http://java.sun.com/products/jdk/1.3/docs/guide/reflection/index.html>.
- [12] J. Peterson, C. Elliott, and G. S. Ling. *Fran 1.1 Users Manual*. Dept. of Computer Science, Yale University, June 1998. Included in Fran distribution available at <http://www.research.microsoft.com/conal/Fran/>.
- [13] J. Peterson, G. Hager, and P. Hudak. A Language for Declarative Robotic Programming. In *International Conference on Robotics and Automation*, 1999.
- [14] J. Peterson, P. Hudak, and C. Elliott. Lambda in Motion: Controlling Robots with Haskell. In *Fist International Workshop on Practical Aspects of Declarative Languages (PADL)*, January 1999.
- [15] S. Peyton-Jones and J. H. (eds.). Report on the Programming Language Haskell 98: A non-strict, purely functional language. Technical Report YaleU/DCS/RR-1106, Dept. of Computer Science, Yale University, 1999.
- [16] A. Reid, J. Peterson, P. Hudak, and G. Hager. Prototyping real-time vision systems: An experiment in dsl design. In *Proceedings of ICSE 99: Intl. Conf. on Software Engineering*, May 1999.
- [17] M. Sage. *FranTk: A Declarative GUI System for Haskell*. Dept. of Computer Science, University of Glasgow, 1999. Available at <http://www.haskell.org/FranTk/userman.pdf>.
- [18] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

A Implementation of Switcher

```
/**
 * Switcher is the fundamental combinator for converting an Event into a
 * Behavior. Switcher takes a Behavior b and a BehaviorEvent e, and
 * produces a new behavior. The new behavior initially follows b. But on
 * every event occurrence, the switcher 'switches' to following the
 * behavior carried in e.
 */
public class Switcher
    implements FRPEventListener, Behavior, PropertyChangeListener {

    /** behavior we are watching: */
    private Behavior target = null;
    /** Delegate who handles listener lists: */
    PropertyChangeSupport listenerManager;

    /**
     * Construct a new switcher from the given behavior and Event source.
     * @param behavior Behavior to follow initially
     * @param source FRPEventSource. This Event source should produce
     * BehaviorEvent's on each occurrence.
     */
    public Switcher(Behavior behavior,
                    FRPEventSource source) {
        listenerManager = new PropertyChangeSupport(this);
        setTarget(b behavior);
        source.addFRPEventListener(this);
    }

    /** Accessor to read the current value of this Behavior. */
    public Object getValue() {
        // just forward to behavior we are observing:
        return target.getValue();
    }

    /** invoked whenever the target property changes: */
    public void propertyChange(PropertyChangeEvent e) {
        // propagate change from target property by notifying
        // all registered listeners:
        listenerManager.firePropertyChange("value",
                                           e.getOldValue(),
                                           e.getNewValue());
    }

    /** invoked when event occurs on event source */
    public void eventOccured(FRPEvent event) {
        BehaviorEvent be = (BehaviorEvent) event;

        // switch behavior being observed to behavior carried in
        // event occurrence.
        setTarget(be.getBehavior());
    }

    /** Set the target behavior to observe. We first de-register with
     * the current target, and then register with the new target.
     */
    protected void setTarget(Behavior newTarget) {
        if (target != null) {
            target.removePropertyChangeListener(this);
            listenerManager.firePropertyChange("value",
                                              target.getValue(),
                                              newTarget.getValue());
        }
        target = newTarget;
        target.addPropertyChangeListener(this);
    }

    // omitted: addPropertyChangeListener, removePropertyChangeListener
}
```

Figure 4: Typical FRP Combinator implementation

B Example Application Using Frappe

```
/**
 * Display a circle that changes color from red to blue depending on which
 * mouse button is pressed.
 */
public class FranTest5 {
    protected CircleBean circle;
    protected FranFrame frame;

    FranTest5() {
        frame = new FranFrame();
        Franimator franimator = frame.getFraminator();
        Time time = franimator.getTimeInstance();

        try {
            FRPEventSource lbpEventSource =
                FRPUtilities.makeFRPEvent(franimator,
                                         "franMouse",
                                         "lbp");

            FRPEventSource lbpWithRed = new EventBind(lbpEventSource,
                                                       new ConstB(Color.red));

            FRPEventSource rbpEventSource =
                FRPUtilities.makeFRPEvent(franimator,
                                         "franMouse",
                                         "rbp");

            FRPEventSource rbpWithBlue = new EventBind(rbpEventSource,
                                                       new ConstB(Color.blue));

            // now merge the two events streams:
            FRPEventSource mouseColorEvents = new EventMerge(lbpWithRed,
                                                             rbpWithBlue);

            // create a circle at the origin with radius 50:
            circle = new CircleBean(0, 0, 50);

            // ShapeImage is a Bean that renders an image in a
            // target JComponent whenever its "shape" property changes.
            ShapeImage renderer = new ShapeImage(400, 400, franimator);

            // connect circle property to renderer's shape property:
            Behavior circleB = new PropertyObserver(circle, "shape");
            new BehaviorConnector(circleB, renderer, "shape");

            Behavior colorB = new Switcher(new ConstB(Color.blue),
                                           mouseColorEvents);

            // connect colorB to renderer's "fillColor" property:
            new BehaviorConnector(colorB, renderer, "fillColor");

            // finally, obtain a Behavior from renderer, and set this
            // as the ImageB to be rendered:
            Behavior imgB =
                FRPUtilities.makeBehavior(renderer, "image");
            franimator.setImageB(imgB);

        } catch (Exception e) {
            System.err.println("Exception connecting behaviors to properties:");
            e.printStackTrace();
        }

        // and start the animation:
        frame.start();
    }

    public static void main(String args[]) {
        FranTest5 ftest = new FranTest5();
    }
}
```

Figure 5: Example application using Frappe